THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Informatique

Présentée et soutenue par :
Mahamadou Abdoulaye TOURE

le : vendredi 18 juin 2010

Titre :

Administration d'applications réparties à grande échelle

JURY

M. Marc DALMAU, Université de PAU et des Pays de l'Adour, Rapporteur
M. Noël De PALMA, Institut National Polytechnique de Grenoble, Rapporteur
M. Michel DAYDE, l'Institut National Polytechnique de Toulouse, Examinateur
M. Daniel HAGIMONT, Institut National Polytechnique de Toulouse, Directeur de Thèse

---

Ecole doctorale :
Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :
Institut de Recherche en Informatique de Toulouse/ENSEEIHT

Directeur(s) de Thèse :

M. Daniel HAGIMONT

Rapporteurs :
M. Marc DALMAU, Université de PAU et des Pays de l'Adour
M. Noël De PALMA, Institut National Polytechnique de Grenoble

A ma famille...

keep six honest serving-men (They tought me all I knew) : Their names are What and Why and When And How and Where and Who.
[J'ai toujours près de moi six fidèles amis (Ils m'ont appris tout ce que je sais). Leurs noms sont Pourquoi, Quand, Où, Quoi, Comment et Qui.]
Rudyard Kipling, The Serving-men.

Le savoir est l'unique fortune que l'on peut entièrement donner sans en rien diminuer.
Amadou Hampâté BAH.

« ***Il faut toujours remercier l'arbre à karité sous lequel on a ramassé de bons fruits pendant la bonne saison*** ». **Ahmadou Kourouma ; Extrait d'Allah n'est pas obligé.**

Ça y est c'est Le Moment ! Celui qui marque Le Point Final de ces longues années d'études. Les remerciements, par habitude, sont rédigés après la présentation des travaux. Je ne fais pas défaut à la règle.

La thèse est un effort de fond qui ne peut pas être réalisé par une seule personne. Par conséquent je remercierai ici, ceux qui m'ont aidé au cours de ces dernières années de travail. D'avance, merci à ceux que j'aurais oublié.

Je tiens d'abord à remercier tous les membres du jury qui ont accepté d'évaluer mon travail. Un très grand merci à mes rapporteurs, Marc DALMAU et Noël DE PALMA pour le temps qu'ils ont consacré à rapporter cette thèse et toute l'attention et la pertinence qui ont été les leurs. Merci à Michel DAYDE pour avoir accepté de présider le jury et d'examiner cette thèse.

Je remercie aussi mon directeur de thèse Daniel HAGIMONT et mon encadrant technique Laurent BROTO d'avoir été présents tout au long de ces années. Merci à toi Daniel pour ton soutien dans mes travaux, pour ces discussions et ces remarques toujours constructives qui font avancer. Merci à toi Laurent de m'avoir aidé techniquement.

Je remercie également chacun des membres de l'équipe ASTRE. Un remerciement particulier pour les membres de l'équipe ASTRE du pôle ENSEEIHT : Aeiman GADAFI, Suzy TEMATE, Alain TCHANA, Raymond ESSOMBA, Larissa MAYAP, Tran SON pour cette bonne ambiance qui règne au sein de l'équipe. Merci à toi Alain TCHANA pour les bons moments passés ensemble de ton stage de Master jusqu'à la fin de ma thèse.

Je tiens aussi remercier et surtout à souligner la disponibilité, le soutien permanent et la bonne humeur de nos secrétaires exceptionnelles, Les « Sylvie ».

Je tiens enfin à remercier ma famille et mes amis. Merci à mes parents pour leur soutien permanent. Merci à mes deux oncles exceptionnels : Almousstakime (KOWA) TOURE et Abdourhamane (DOUMMA) TOURE sans lesquels je n'aurai jamais pu en arriver là. Merci à mes amis de grin : Amadou BABA BAGAYOGO (BAGA) et Cheick OUMAR HAIDARA (CHEKO). Merci à toi Jacqueline KONATE d'avoir pris de ton temps si précieux pour préparer le pot de cette thèse. Merci à Kadidia TOURE et à la famille BA pour les bons moments passés ensemble à Toulouse. Merci à toi Aboubacar DIALLO (ABA) pour les séances ciné de vendredis ! Enfin merci à toi Hadiaratou I TOURE de m'avoir soutenu, aidé moralement malgré la distance qui nous sépare.

Et je ne pourrai terminer cette page sans avoir une pensée très émue pour ceux qui nous ont quittés beaucoup trop tôt, et en particulier, pour mes deux tantes que j'appelle intimement ATCHA et ZARHA ... Si je suis là aujourd'hui, c'est aussi grâce à vous ...

L'administration d'une application est une tâche de plus en plus complexe et coûteuse en ressources humaines et matérielles. Nous nous intéressons dans cette thèse à l'administration dans un contexte de grande échelle. Dans ce contexte particulier, nous disposons généralement de plusieurs entités logicielles qui doivent être déployées et gérées sur une infrastructure matérielle de type grille composée de nombreuses machines géographiquement dispersées. L'administration sur ce type d'infrastructure pose de multiples problèmes ***d'expressivité*** *liés à la description des éléments à administrer, de* ***performance*** *liés à la charge des processus d'administration et la répartition géographique des sites de la grille,* ***d'hétérogénéité*** *matérielle et logicielle, et de* ***dynamicité*** *(panne, coupure de lien réseau, etc.). Nos contributions portent essentiellement sur les problèmes précédemment cités. Un formalisme de description tenant compte du facteur d'échelle est proposé pour décrire l'infrastructure matérielle et logicielle. Nous proposons la répartition de la charge et la diminution du coût de l'administration en utilisant plusieurs systèmes d'administration et en personnalisant la phase d'installation du déploiement. Enfin nous proposons une gestion de l'hétérogénéité matérielle et logicielle. Le travail de cette thèse s'inscrit dans le cadre du projet* ***TUNe****. Nous proposons donc une application et une implantation de ces contributions au système TUNe afin de valider notre approche dans le cadre d'une expérimentation en vraie grandeur...*

Administration of distributed systems is a task increasingly complex and expensive. It consists in to carry out two main activities : deployment and management of application in the process of running. The activity of the deployment is subdivided into several activities : description of hardware and software, configuration, installation and starting the application. This thesis work focuses on large-scale administration which consists to deploy and manage a distributed legacy application composed of several thousands of software entities on physical infrastructure grid made up of hundreds or thousands of machines. The administration of this type of infrastructure creates many problems of expressiveness, performance, heterogeneity and dynamicity (breakdown of machine, network, ...). These problems are generally caused to the scale and geographical distribution of the sites (set of clusters). This thesis contributes to resolve the problems previously cited. Therefore, we propose higher-level descriptions formalisms to describe the structure of hardware and software infrastructures. To reduce the load and increase the performance of the administration, we propose to distribute the deployment system in a hierarchical way in order to distribute the load. The work of this thesis comes the scope of the ***TUNe*** *(autonomic management system) project. Therefore, we propose to hierarchize TUNe in order to adapt in the context of large-scale administration. We show how to describe the hierarchy of systems. We also show how to take into account the specificity of the hardware infrastructure at the time of deployment notably the topology, characteristics and types of machine. We define a process langage allowing to describe the process installation which allow managers to define thier own installation constraints, according to their needs and preferences. We explore the management of heterogeneity during deployment. Finaly our prototype is validated by an implementation and in the context of a real experimentation.*

Table des matières

Chapitre 1

Introduction

1.1 Introduction

Le calcul haute performance consiste à résoudre des problèmes nécessitant de grandes capacités de calcul. Ce sont par exemple les simulations cosmologiques, les prédictions météorologiques, la détermination de formes de protéines, etc. Historiquement ces calculs ont été réalisés sur des ordinateurs massivement parallèles, puis l'émergence des réseaux hautes performances a permis l'agrégation de ressources de calcul homogènes en grappes, puis de ressources hétérogènes fortement distribuées en grilles.

Une *grille* est formée d'un très grand nombre de ressources de calculs et de stockages. Le déploiement de réseau à très haut débit dans l'internet a permis la construction de ce genre d'infrastructures informatiques distribuées. Théoriquement, au sein d'une grille informatique, il est possible d'accéder à la puissance de calcul de manière analogue à la puissance électrique. En effet le terme *grille* (*grid* en anglais) provient de l'analogie avec les réseaux électriques (*electrical power grid*) qui produisent sans cesse de l'énergie électrique qui est fournie, à la demande, à l'usager qui en a besoin. L'idée des grilles est semblable à cela à la différence que les ressources fournies aux consommateurs ne sont plus de l'électricité mais de la puissance de calcul, de l'espace de stockage informatique, etc. Cette analogie signifie qu'une grille doit être aussi facile à utiliser qu'un réseau électrique. L'infrastructure grille est de nature hétérogène tant du point de vue machines que du point de vue réseaux de communication, dispersée géographiquement dans les sites éloignés, dynamique (des machines qui disparaissent pour des raisons de pannes ou apparaissent régulièrement). La complexité d'utilisation d'une telle infrastructure nous ramène à répondre à une question : **comment administrer une application répartie sur une infrastructure de type grille ?**

Administrer une application revient à la déployer sur une infrastructure matérielle et se charger de sa gestion en cours d'exécution. Le processus de déploiement

d'une application s'étend de sa description jusqu'à son exécution effective et sa terminaison. La description de l'application consiste à décrire, dans un formalisme convenu, les paramètres de configuration de chaque entité logicielle de l'application ainsi que son architecture globale. La gestion d'une application permet de la modifier en cours d'exécution ou d'effectuer des opérations de maintenance pour satisfaire les besoins de leurs utilisateurs ou pour prendre en compte la modification de leur environnement d'exécution. Ces différents processus s'avèrent difficiles notamment dans un contexte de grande échelle et peuvent engendrer des multiples problèmes.

Les problèmes de l'administration d'applications réparties sur une grille de machines proviennent en grande partie des caractéristiques même de la grille. En effet l'utilisation de la grille s'avère être compliquée, même pour les spécialistes du domaine. Ceci est dû à l'échelle car une grille est composée de plusieurs milliers de ressources hétérogènes et réparties au sein de sites géographiquement distants. Ces sites font partie de multiples organisations, empêchant toute gestion des ressources à l'échelle globale. Administrer des applications distribuées sur de telles infrastructures requiert beaucoup d'efforts de la part des administrateurs pour découvrir et sélectionner les ressources, installer et configurer les programmes, transférer les données, lancer les calculs, surveiller l'exécution puis, enfin, récupérer les résultats. Aujourd'hui, les applications réparties sont encore principalement administrées manuellement. Des outils pour simplifier le processus d'administration existent [LPP05], [CCD06], [FDDM08], mais restent incomplets : certains sont spécifiques à un type d'application [LPP05], [CCD06] et effectuent l'administration de façon centralisée (toutes les opérations d'administration sont exécutée à partir d'une seule machine), d'autres effectuent seulement du déploiement [FDDM08]. Les grilles étant des environnements complexes, dynamiques et hétérogènes, il est nécessaire de proposer des outils simples et efficaces permettant de simplifier l'administration des applications à ses utilisateurs. Cette thèse s'inscrit dans le cadre du projet **TUNe** qui propose un formalisme de haut niveau pour la spécification des politiques d'administration autonome. Dans cette thèse, nous nous intéressons à la mise en œuvre d'un système tel que TUNe dans un contexte grille.

Ce document est structuré de la manière suivante. Le **chapitre 2** dresse, après avoir présenté la définition des termes et les notions utilisées dans ce manuscrit, les problématiques d'administration d'applications réparties dans un contexte de grande échelle. Le **chapitre 3** introduit le contexte de notre étude. Il présente les applications utilisées pour les expérimentations (*DIET* et *LogService*) ainsi que l'environnement d'administration utilisé pour la mise en œuvre des contributions de cette thèse. **Le chapitre 4** brosse un tour d'horizon des différents outils de déploiement et d'administration autonome (***État de l'art***). A partir de cette étude de l'existant, nous identifions les lacunes à combler dans ces outils. Dans les **chapitres (5, 6 et 7)**, nous présentons nos différentes contributions portant sur *l'expressivité*, la *performance* et la *gestion de l'hétérogénéité*. Nous expliquons le concept général de chaque contribution puis nous présentons une mise en œuvre et une implantation dans le système d'administration *TUNe*. Le **chapitre 8** présente la validation

des contributions. Cette validation passe par des expérimentations en utilisant une infrastructure de type grille *Grid'5000*. Enfin le dernier **chapitre 9** présente une conclusion qui retrace les points essentiels de la thèse, et présente plusieurs pistes de recherches prospectives dans le prolongement de nos travaux.

Première partie

Problématique et contexte

Chapitre 2

Position du problème

Table des matières

2.1 Contexte d'étude

2.1.1 Définitions

Pour supprimer toute ambiguïté sur les termes que nous utilisons dans ce document, nous les définissons dans cette section.

Définition 2.1.1 ***: Nœud*** - *C'est une machine physique permettant d'exécuter des tâches. Cela peut être une station de travail, ou un simple PC. Un nœud est traditionnellement utilisé à l'aide d'un système d'exploitation qu'il exécute.*

Définition 2.1.2 ***: Tâche*** - *Une tâche est une suite d'opérations qui résultent de l'exécution d'un programme informatique.*

Définition 2.1.3 ***: Entité logicielle*** - *Une entité logicielle est un élément de base d'une application. C'est une unité logicielle offrant des services spécifiques d'une application. Par exemple pour une application J2EE, Tomcat, MySQL sont des entités logicielles.*

Définition 2.1.4 ***: Application patrimoniale*** - *Une application est dite patrimoniale lorsqu'elle n'est disponible que sous forme binaire. Le code source de l'application n'est pas accessible.*

Définition 2.1.5 ***: Ressource*** - *Une ressource peut être définie comme les moyens dont dispose un nœud pour effectuer les calculs. Les processeurs, la quantité de mémoire, les cartes réseau sont considérés comme des ressources d'un nœud.*

2.1.2 Les environnements informatiques

Les travaux de recherche scientifiques s'intéressent de plus en plus aux grilles informatiques. Les orientations différent selon le domaine. Chaque domaine a sa propre définition. Nous commençons dans cette section, par définir la notion du *cluster* qui est un ensemble de machines homogènes localisées généralement dans une même localité. A partir de cette définition, nous présentons la notion des *grilles* qui est une généralisation de la notion de *cluster*.

2.1.2.1 Cluster

Le terme de *cluster* signifie grappe en français. Dans sa forme la plus simple, un *cluster* est un ensemble de deux ordinateurs ou plus, appelés nœud, qui travaillent ensemble pour fournir un service. Le concept de *cluster* est apparu pour résoudre les problèmes nécessitant une capacité de calculs de plusieurs machines. Il permet d'assembler la puissance des ordinateurs pour diviser le temps d'exécution d'un programme ou, à travers la redondance, obtenir plus grande fiabilité. Un *cluster* utilise donc plusieurs machines pour fournir un environnement informatique plus performant en vu de réaliser des calculs parallèles ou distribués [Buy99].

On sein d'un *cluster*, on peut distinguer plusieurs types de nœuds :

– *Nœud de calculs* : C'est un nœud qui est chargé de l'exécution des tâches au sein du *cluster*. Le choix du type de matériel est très important pour ce type de nœud. Les performances dépendent bien évidemment du type de processeurs et de la quantité mémoire, mais aussi de la carte réseau ;

- *Nœud utilisateur* : Les nœuds d'un *cluster* sont généralement sur un sous réseau privé qui ne peut être accédé directement depuis l'extérieur pour des raisons de sécurité. Un nœud utilisateur permet de fournir aux utilisateurs un accès (qui peut être externe) aux nœuds de calculs du cluster pour soumettre un nouveau job (exécution d'une application, d'un script etc.) ou récupérer les résultats d'une tâche précédemment effectuée ;
- *Nœud de management* : Ce type de nœud est chargé de la gestion des autres nœuds. Il permet de modifier leur configuration, de corriger un problème. Il gère les alarmes ou les évènements provenant du cluster ;
- *Nœud d'installation* : Les nœuds de calculs peuvent être régulièrement mis à jour, reconfigurés ou réinstallés. Un nœud d'installation permet de faire cette tâche de manière simple en fournissant une image unique à tous les nœuds ;
- *Nœud de contrôle* : Le nœud de contrôle fournit les services nécessaires aux autres nœuds pour assurer leur cohésion. Il joue généralement le rôle de serveur DHCP ou NFS au sein du cluster.

La notion de *cluster* peut être étendue pour donner des sites. Un site est formé par un ou plusieurs clusters reliés par un réseau de communication (par exemple un LAN) qui sont localisés géographiquement dans le même institut, campus universitaire, centre de calcul, entreprise ou chez un individu, et qui forment un domaine d'administration commun, uniforme et coordonné.

2.1.2.2 Grille informatique

Le terme *Grille* [LPP04b] [KF98b] a été introduit pour la première fois aux Etat Unis durant les années 1990 pour décrire une infrastructure de calcul distribué, utilisée dans les projets de recherche scientifiques et industriels [Kie04] [GTLAG05]. Une grille mutualise un ensemble de machines géographiquement distribuées sur plusieurs sites. Un site peut être vu comme un ensemble de clusters, composé d'un ensemble de machines situées généralement à la même localité et qui forment un domaine d'administration local, uniforme et coordonné.

La notion de grille s'inspire fortement de la grille d'électricité (Power Grid). En effet, une grille peut être vue comme un instrument qui fournit de la puissance de calculs et/ou de la capacité de stockage de la même manière que le réseau électrique fournit de la puissance électrique. La vision des inventeurs de ce terme est qu'il sera possible, à terme, de se *brancher* sur une grille informatique pour obtenir de la puissance de calcul et/ou de stockage de données sans savoir ni où ni comment cette puissance est fournie, à l'image de ce qui se passe pour l'électricité. Cependant la mise en œuvre de cette transparence n'est pas triviale vue les caractéristiques spécifiques aux grilles.

Une grille est caractérisée par sa répartition sur différents sites qui ne sont pas sous administration commune. Cela conduit à une grande hétérogénéité tant au niveau matériel que de l'environnement logiciel. Chaque site d'une grille admet sa

propre politique d'administration, son propre protocole d'accès et d'authentification. Les politiques de sécurité peuvent aussi être différentes d'un site à l'autre. Les sites ne partagent pas non plus un même système de fichiers. La grille n'a pas une structure statique. Que ce soit du fait de pannes matérielles, de remplacements ou d'ajouts, des ressources peuvent apparaître ou disparaître à tout instant. Une grille peut comporter une grande variété de technologies d'interconnexion réseau, et toute une hiérarchie de réseaux en termes d'étendue géographique, et en termes de performances des communications (débit, latence, etc.). Des réseaux longue distance (Wide-Area Network,WAN) relient les sites de la grille. Les nœuds à l'intérieur de chaque site peuvent être inter-connectés par des réseaux locaux (LAN) ou par des réseaux haute performance (SAN) au sein d'un cluster. La figure 2.1 montre un exemple d'une grille pour l'infrastructure *Grid5000* [CCD+05a].

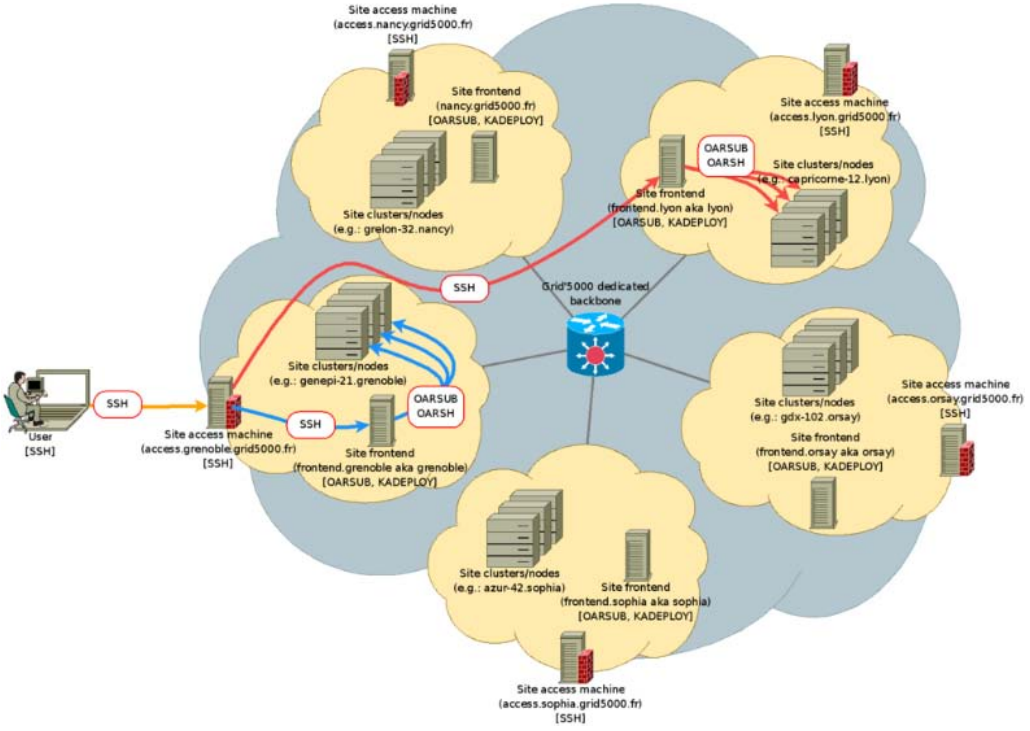

FIGURE 2.1 – Exemple d'une grille informatique (Grid5000)

Il existe de nombreux travaux autour des grilles et des environnements distribués à grande échelle. La majorité [ADZ00], [FFG+01], [JCC+03], [CIG+03] traite plutôt le déploiement ou l'aspect gestion de ressources. Dans ce qui suit, nous nous intéressons aux différents problèmes liés à l'administration d'applications réparties à grande échelle. Ces problèmes sont généralement liés aux facteurs d'échelles.

2.2 Administration d'un environnement informatique

2.2.1 Définition générale de l'administration

L'administration d'un environnement informatique consiste à gérer les infrastructures matérielles et logicielles de cet environnement. Selon [Kle88], les fonctions d'administrations sont composées de plusieurs sous fonctions d'administration :

- *La gestion du déploiement* : Cette fonction est subdivisée en plusieurs tâches à savoir : la *configuration*, *l'installation*, le *démarrage*, la *désinstallation* et l'*arrêt* d'un logiciel. Nous reviendrons plus en détails sur le déploiement dans cette section ;
- *La gestion de performance* : Cette fonction consiste à assurer une certaine qualité de service. L'environnement à administrer doit être adapté aux différentes variations de performance et aux pics de charge. Cela peut être, par exemple dans le cas d'un serveur web, la diminution du temps de réponse du serveur ou l'augmentation du nombre de clients pouvant être connectés ;
- *La gestion des pannes* : Les pannes sont très fréquentes dans un environnement composé de centaines voire de milliers de machines et d'entités logicielles. Il existe deux types de panne : panne matérielle et panne logicielle. La gestion des pannes consiste à détecter la défaillance d'une machine ou d'un logiciel en vue de permettre aux administrateurs de ramener le système dans un état opérationnel. La gestion des pannes est une tâche primordiale dans l'administration d'un système logiciel ;
- *La gestion de la sécurité* : Les utilisateurs d'un environnement informatique n'ont pas les mêmes droits d'accès aux ressources installées. Certaines données sont sensibles et ne doivent pas être accessibles à tout le monde. Il est donc nécessaire de donner un droit d'accès à chaque utilisateur et mettre des mécanismes de vérification d'accès aux ressources.

2.2.2 L'administration autonome

L'administration autonome est une approche proposée par IBM [Hor01] [Mur04] [Kep05] afin de donner la possibilité à un système de s'auto-administrer c'est-à-dire administrer lui-même les éléments qui le composent sans intervention humaine afin d'assurer son bon fonctionnement. Cela inclut l'*auto-configuration* (configuration automatique suivant des règles prédéfinies), l'*auto-optimisation* (le système détecte les problèmes de performance et entreprend lui-même les mesures nécessaires pour y remédier), l'*auto-réparation* (le système est capable de détecter et réparer la panne d'un ou plusieurs de ses éléments) et l'*auto-protection* (prise des mesures nécessaires pour se protéger des attaques malveillantes et savoir se défendre contre ces

attaques). Ainsi on parle de *systèmes autonomes* c'est-à-dire de systèmes capables de s'auto-gérer. L'objectif de cette approche est de diminuer l'intervention d'un administrateur, éventuellement le remplacer partout ou cela est possible. Le rôle de l'administrateur est alors réduit à la définition des politiques d'administration qu'il souhaite voir appliquées au système dont il a la charge, tandis que le système autonome prend lui-même en charge la mise en application et le maintien de ces politiques, quoi qu'il arrive et sans nécessiter une intervention humaine.

Notre objectif est de fournir des solutions permettant de passer à l'échelle lors de l'administration. Dans la suite de cette section, nous présentons plus en détail l'activité du déploiement et la reconfiguration.

2.2.3 Déploiement d'une application

Le déploiement est un ensemble d'activités faisant partie du cycle de vie d'une application [CFH+98]. Cet ensemble d'activités concerne : l'*allocation des ressources* nécessaires au déploiement, la *description* des différents paramètres de configuration ainsi que l'architecture logicielle de l'application à déployer, *l'installation*, la *configuration*, le *démarrage*, la *désinstallation* de l'application et l'*arrêt* du processus de déploiement. Nous présentons plus en détaille chacune de ces activités :

- *Allocation des ressources* : Cette activité consiste à réserver les machines nécessaires à l'exécution de l'application à déployer en utilisant des outils spécifiques à l'infrastructure matérielle. Certaines applications peuvent nécessiter des machines bien particulières au niveau de la puissance de calcul, la quantité de mémoire disponible, etc. Pour cela, des critères peuvent être précisés lors de la réservation. Cette réservation peut également être effectuée sans critère particulier c'est-à-dire en précisant juste le nombre de machine nécessaire. Il existe des outils de réservation notamment *OAR* [CCG+05] pour les infrastructures matérielles de type grappes et *OARGRID* [OAR0x] pour les grilles ;
- *Description* : La description d'une application consiste à exprimer dans un langage, les configurations des différentes entités logicielles ainsi que l'architecture logicielle de l'application ;
- *Installation* : La phase d'installation du déploiement d'une application a pour objectif de préparer l'environnement de l'application. Elle permet de mettre en place les ressources nécessaires au fonctionnement de l'application (transfert des paquetages, des libraires, etc.) ;
- *Le démarrage* : Cette activité consiste à lancer l'application sur les machines physiques. Par ailleurs, il peut exister une dépendance de démarrage entre les entités logicielles de l'application. Il faut à cet effet pouvoir exprimer, dans un formalisme, l'enchainement des actions de démarrage afin de respecter cette dépendance ;

– *Désinstallation* : Elle correspond à l'opération inverse d'installation. Elle consiste à supprimer toutes les traces laissées par l'activité d'installation (suppression des fichiers, des répertoires, etc.) ;
– *Arrêt* : Cette dernière étape correspond à l'arrêt du système de déploiement. Cela peut être effectué au fur et à mesure qu'on désinstalle l'application. L'étape se termine généralement lorsque le système de déploiement rend la main au système d'exploitation qui l'a démarré.

Dans le cas d'une application patrimoniale, la phase d'installation du déploiement est subdivisée en plusieurs sous activités. Elle commence par la mise en place des ressources logicielles sur les machines. Cela peut être par exemple la création des répertoires d'installation,transfert des paquetages et éventuellement la génération des fichiers de configuration de l'application. L'installation se poursuit par le démarrage des différentes entités logicielles de l'application (voir figure 2.2).

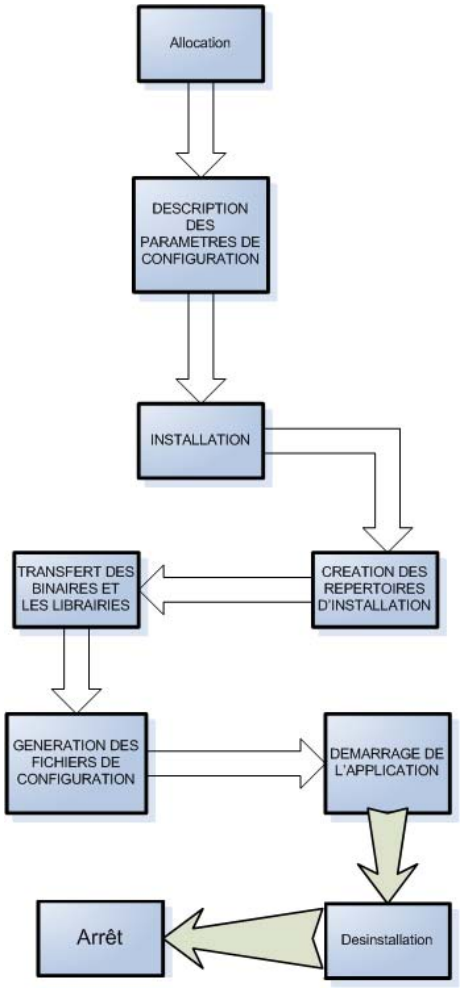

FIGURE 2.2 – Processus de déploiement d'une application patrimoniale

2.2.4 Reconfiguration

La reconfiguration fait référence aux différentes modifications que peut subir une application en cours d'exécution. Cela peut être la suppression, l'ajout ou le remplacement d'une entité logicielle, la réparation d'une panne ou simplement le changement des paramètres de configuration. Nous distinguons deux types de reconfiguration :

- *La reconfiguration non fonctionnelle* : Elle peut consister à modifier les paramètres de configuration d'une entité logicielle afin d'ajuster son comportement convenablement. Il peut s'agir de la modification de l'architecture de l'application sans modifier sa fonction initiale. Ce type de reconfiguration peut s'imposer par exemple lors du changement de contexte de l'application. En effet le contexte d'une application peut changer au fil du temps. Certains de ces changements peuvent rendre l'application dans un état qui ne répond plus au cahier des charges. Pour cette raison, l'application doit être reconfigurée pour prendre en compte les nouvelles conditions de son contexte d'exécution. Il faut par ailleurs noter qu'une reconfiguration non fonctionnelle peut nécessiter l'arrêt et le redémarrage partiel ou total de l'application selon l'impact de l'opération de reconfiguration effectuée ;
- *La reconfiguration fonctionnelle*. Elle correspond à la modification de la structure interne du logiciel par l'ajout/suppression d'une nouvelle entité ou la modification des dépendances entre ces entités. Il peut s'agir, dans le cas des applications à composant, d'une modification de l'implantation d'un composant, ou la modification de l'interface d'un composant, etc. La conséquence d'une reconfiguration fonctionnelle est la modification de la fonction de l'application (étendue ou diminuée). Ce type de reconfiguration peut nécessiter plusieurs opérations. Par exemple pour l'ajout d'une nouvelle entité logicielle, il faut la créer, la configurer, mettre à jour les données et les dépendances et enfin la démarrer.

Le rôle de la reconfiguration est d'augmenter la qualité des services d'une application, faire évoluer, adapter une application en tenant compte des changements au niveau de l'infrastructure matérielle ou logicielle. Dans ce manuscrit, nous nous intéressons au type de reconfiguration non fonctionnelle.

2.3 Problématique

L'usage des grilles informatiques est indispensable pour la résolution des problèmes qui demandent beaucoup plus de capacités de calcul. Cet usage passe par l'utilisation des applications réparties telles que les applications scientifiques qui nécessitent de l'administration (installation, configuration, démarrage, reconfiguration, etc.).

Construire une grille de calcul, revient à faire en sorte que des machines, de nature hétérogène, situées dans des sites distants puissent apparaître comme une unique ressource. Au delà de la simple connexion réseau nécessaire pour relier ces machines entre elles il faut rajouter un système dont le rôle est d'assurer l'administration des applications utilisant cette grille. Ce système a pour but de simplifier et d'automatiser les différentes tâches d'administration des applications. Administrer une application sur une infrastructure aussi complexe n'est pas une tâche facile. Il faut en effet déployer (installer, configurer, démarrer les entités logicielles) et se charger de la gestion de l'application en cours d'exécution (la réparation des pannes, la gestion des variations de performances, la gestion des ressources et la gestion de la sécurité).

L'administration fait l'objet de nombreux projets de recherche et engendre plusieurs problèmes que nous allons décrire dans cette section. Ces problèmes sont généralement liés à deux facteurs :

- **Le facteur d'échelle** : Ici la notion de facteur d'échelle recouvre à la fois le nombre d'entités logicielles à administrer et la dimension du réseau c'est-à-dire le nombre de nœuds caractérisant les clusters de la grille. Administrer de centaines voire de milliers de logiciels et de nœuds pose généralement de problèmes d'échelle. La problématique liée au facteur d'échelle est subdivisée en trois sous problèmes : la *performance*, *l'hétérogénéité* et *l'expressivité*. En effet administrer des multiples entités logicielles et matérielles à partir d'une seule machine peut avoir un effet sur la *performance* de l'administration (temps de déploiement, temps de réaction, etc.). La grille est de nature hétérogène au niveau réseau, matériel (architecture processeur, des machines de 32 ou 64 bits, etc.) et logiciel. Cette *hétérogénéité* engendre des problèmes car il faut déployer pour chaque machine la version de l'application qui lui est compatible selon ses caractéristiques matérielles et logicielles. Le déploiement d'une application nécessite la description de son architecture et ses paramètres de configuration ainsi que l'infrastructure matérielle sur laquelle on l'administre. Le problème *d'expressivité* est lié au formalisme proposé pour effectuer cette description ;
- **Le facteur Dynamique** : Une grille est caractérisée par sa dynamicité : panne de machine, coupure de lien de réseau, modification de la charge, variation de performance, etc. Nous distinguons plusieurs types de pannes : la panne matérielle liée aux ressources matérielles de la grille ; la panne logicielle liée aux applications installées sur les nœuds ; la déconnexion d'une machine ; etc.

2.3.1 Facteur d'échelle

Nous présentons dans cette section la problématique liée au facteur d'échelle. Comme évoqué précédemment, cette problématique est subdivisée en trois sous problèmes : la *performance*, l'*hétérogénéité* et l'*expressivité*.

2.3.1.1 Performance

L'administration d'une application répartie à grande échelle est difficile compte tenu de la dimension de l'application en terme du nombre d'entités logicielles qui la compose, et le nombre de nœuds de l'infrastructure matérielle grille. En effet une application répartie à grande échelle est composée de centaines voire de milliers d'entités logicielles. Une grille est constituée de plusieurs milliers ou plusieurs dizaines de milliers de nœuds qui peuvent être répartis dans plusieurs régions et pays. La difficulté de l'administration est due à cette quantité d'information qui augmente le nombre de tâches d'administration à effectuer. Une tâche d'administration peut être par exemple la copie des paquetages sur les nœuds, la configuration, le démarrage d'une entité logicielle, etc. L'exécution d'une tâche d'administration nécessite généralement la création de plusieurs processus sur la machine locale (qui exécute les tâches d'administration) et distante (sur laquelle l'entité logicielle est administrée). Chaque tâche d'administration a un coût et représente une charge pour la machine qui administre. Cette charge peut avoir une influence sur la performance de l'administration (temps de déploiement, temps de réactivité, etc.)

Pour savoir la provenance de la charge de l'administration, nous allons examiner les tâches du déploiement lors de l'administration d'une application. En effet le processus de déploiement fait référence à une multitude de tâches notamment l'installation (préparation de l'environnement matériel, transfert des fichiers), la configuration, le démarrage de l'application. Ces tâches peuvent demander beaucoup de ressources en terme de puissance de processeur ou de la quantité de mémoire. Le transfert des fichiers ainsi que le démarrage de l'application (création de processus permettant le démarrage d'une entité logicielle) nécessitent de nombreuses connexions sur les nœuds distants. Ces différentes connexions représentent une lourde charge pour le nœud qui administre. Pour des raisons évoquées précédemment, effectuer l'administration à partir d'une seule machine peut poser de problèmes lors du passage à l'échelle. L'utilisation de cette approche qu'on appelle *approche centralisée* peut dégrader la performance du système d'administration.

En résumé, l'administration des applications réparties sur un grand nombre de nœuds géographiquement dispersés peut s'avérer coûteuse en ressources. Beaucoup d'approches reposent sur une approche centralisée où le passage à l'échelle est compromis par le risque de limiter la performance en terme du temps d'exécution des différentes tâches d'administration et le temps de réaction en réponse à des événements. Pour des raisons de performances, les systèmes d'administration doivent être décentralisés afin que les tâches d'administration soient exécutées à partir de plusieurs machines.

2.3.1.2 Hétérogénéité

L'hétérogénéité d'une grille peut être située sur plusieurs niveaux. Le premier niveau est la puissance de calcul. En termes de puissance de calcul, on peut aussi bien trouver des mono processeurs, multiprocesseurs que des ordinateurs de bureau (PC), des serveurs d'exécution, des stations de travail, etc. Le deuxième niveau est l'architecture des machines. En termes d'architecture matérielle, les nœuds peuvent être équipés de différents types de processeurs : PowerPC, compatibles i386, des Alphas, des Mips, etc. Le troisième niveau est l'hétérogénéité des logiciels. En termes d'installation logicielle, les nœuds peuvent avoir différents systèmes d'exploitation avec une version précise (AIX 5.3, IRIX 6.5, Solaris 9, Linux 2.6.10, Windows XP, Windows vista etc.). Les logiciels disponibles et leurs versions peuvent également être différents et installés à des endroits variés (compilateurs, bibliothèques de calcul, etc.). Le quatrième niveau est le réseaux. En termes de réseaux d'interconnexion entre les machines, les liens de communication peuvent avoir des débits, latences, taux de pertes différents. La topologie du réseau peut être différente d'un cluster à un autre. Des clusters de la grille peuvent avoir des serveurs NFS/SMABA installés afin de partager les données entre les nœuds. Enfin le dernier niveau est l'hétérogénéité des politiques d'accès aux ressources. Chaque site d'une grille est administré de façon locale et indépendante avec des politiques d'authentification différentes (TELNET, SSH, etc.), et éventuellement des algorithmes de chiffrement différents.

L'administration d'une application passe par une phase de déploiement dans laquelle des paquetages logiciels sont sélectionnés et installés sur les machines. Ces paquetages peuvent exister en plusieurs versions dépendantes du système d'exploitation, du type de processeur, de la quantité de mémoire disponible, de la carte réseau ou encore du support de stockage, etc. Compte tenu de l'hétérogénéité des machines et les paquetages, le système d'administration doit sélectionner pour chaque machine, la version du paquetage de l'application qu'il lui est compatible. Par ailleurs, l'administrateur peut imposer des contraintes sur les machines à utiliser pour administrer son application lors de cette sélection. En effet un administrateur peut vouloir à ce qu'une entité logicielle soit administrée sur des nœuds ayant une puissance de calcul élevée, avec une quantité de mémoire donnée. Il peut aussi exiger un nombre minimum de processeurs, ou demander la proximité (le nœud le plus proche en terme de connexion réseau) sans autant spécifier les noms des machines.

2.3.1.3 Expressivité

L'une des problématiques de l'administration est la description de l'infrastructure matérielle et logicielle. Cette description est d'autant plus complexe que l'administration s'effectue dans un contexte de grande échelle. En effet dans un contexte de grande échelle, le nombre d'entités logicielles et de machines à décrire est multiple. L'objectif ici est de proposer un langage de description basé sur un formalisme commode, intuitif et adapté tenant compte de ce facteur d'échelle. La reconfiguration

d'une application peut nécessiter un formalisme de description des actions à entreprendre et leurs enchainements. Ce formalisme doit également tenir compte du facteur d'échelle.

Les langages de description d'une application ont pour objectif de définir les paramètres de configuration, les dépendances et l'architecture logicielle de l'application. Ils sont basés sur des concepts tels que les composants qui représentent les entités logicielles d'une application ; les connecteurs qui définissent les types d'interaction entre les entités.

Les langages de description de l'infrastructure matérielle grille ont pour but de décrire les caractéristiques matérielles et logicielles des machines ainsi que la topologie réseau de la grille. Parmi les caractéristiques, on peut citer : le nombre et type de processeur, espace disque libre, mémoire vive, les serveurs installés, le protocole d'accès aux machines, etc. La topologie de la grille donne sa vue structurelle en terme de connexion réseau ainsi que les relations entre les clusters qui la compose.

La définition d'un langage de reconfiguration permet de séparer la logique de reconfiguration de sa mise en œuvre. Ce langage décrit les actions à exécuter pour reconfigurer une application et doit être à la fois facile d'utilisation et générique. Cette facilité peut passer par la réduction des possibilités de reconfiguration en limitant les symboles du langage, ou bien au contraire d'étendre le langage en autorisant l'introspection de l'environnement et la création de nouveaux symboles.

2.3.2 Facteur dynamique

La grande échelle induit inéluctablement de nombreux ajouts ou retraits de nœuds. Ce comportement dynamique peut être dû à des défaillances des nœuds [MCBS03] [LGWT06] ou à des ruptures de liens réseaux. Il peut également être dû à des déconnexions volontaires, dans le cas par exemple où un administrateur souhaite effectuer une mise à jour du système d'exploitation ou encore dans le cas où les nœuds sont partagés seulement une partie de la journée comme dans les réseaux de stations de travail.

Dans le cas d'un application à grande échelle, l'application peut potentiellement être victime de la défaillance d'un nombre arbitraire d'entités et de nœuds, d'autant plus que le nombre de nœuds est très élevé, la probabilité de déconnexion des machines, coupure des liens réseau ou de panne des machines est non négligeable. Ce comportement dynamique doit être pris en compte par les systèmes d'administration.

Face à cette problématique, une approche prometteuse consiste à fournir un système d'administration autonome permettant de gérer automatiquement les tâches d'administration. Ainsi les pannes matérielles et logicielles sont automatiquement détectées et réparées. En effet l'administration autonome apporte une meilleure ré-

activité en effectuant des opérations d'administration (réglage, optimisation, réparation) dynamiquement, en réponse à des observations et sans intervention humaine.

Parmi les différents aspects que recouvre l'administration, un élément central et indispensable est la réparation des pannes. L'objectif est de fournir aux systèmes d'administration la capacité de réparer les pannes matérielles et logicielles. Le système d'administration doit pouvoir, d'une part détecter les défaillances et, d'autre part, décider d'une action à entreprendre pour ramener le fonctionnement à un mode normal. Enfin, il doit disposer de moyens d'action sur l'application administrée afin d'agir sur son état aux vues de son diagnostic. Dans le cas d'une administration autonome, cette réparation des pannes est effectuée de façon autonome.

Ce chapitre a défini ce que nous entendons par cluster, grilles de calcul et administration (une définition générale et l'administration autonome), et a mis en exergue la problématique d'une administration d'applications réparties à grande échelle. Parmi les multiples définitions possibles d'une grille de calcul, nous avons choisi de retenir celle qui la présente comme un ensemble des sites géographiquement dispersés et qui sont pas sous administration commune. Les principales caractéristiques des grilles de calcul sont :

- *l'hétérogénéité* des ressources matérielles (architectures et puissance des ordinateurs, topologies et performances des réseaux de communication, etc.) et logicielles (systèmes d'exploitation, bibliothèques installées, etc.) ;
- *l'échelle* tant en terme de distribution géographique (les sites d'une grille sont localisés d'un pays à un autre, d'un continent à un autre) qu'en terme du nombre de nœuds ;
- *structure dynamique*, la grille n'a pas une structure statique. Que ce soit du fait de pannes matérielles, de remplacements ou d'ajouts, des nœuds peuvent apparaître ou disparaître à tout instant ;
- *manque d'administration commune* : contrairement aux clusters, une grille de calcul n'a pas une seule politique d'administration. Il faut en effet fédérer les différentes politiques d'administration des sites qui la constitue.

La première problématique de l'administration d'applications réparties à grande échelle que nous avons abordée est liée aux caractéristiques de la grille en occurrence le facteur d'échelle (performance, hétérogénéité, expressivité). La seconde problématique est liée au facteur dynamique. Il s'agit ici des pannes matérielles et logicielles qui sont très fréquentes dans le contexte de grille de calcul. Cette problématique nous a montré la nécessité de l'administration autonome qui permet de détecter automatiquement une défaillance et de la réparer sans intervention humaine.

Chapitre 3

Contexte

Table des matières

3.1 Contexte applicatif

Dans ce chapitre, nous présentons deux applications *DIET* et *LogServce* utilisées pour nos expérimentations. Ensuite nous présentons le projet *TUNe* dans lequel nous avons implanté les contributions de cette thèse. *DIET* est un ordonnanceur composé d'un ensemble hiérarchique d'agents pour l'élaboration d'applications basées sur des serveurs de calculs. *LogService* est un logiciel de génération de fichier de traces (fichier de log) développé pour un environnement distribué tel que DIET. Ce logiciel que nous avons utilisé pour détecter les pannes des agents DIET, permet de suivre, par échange de fichiers texte, les traces d'exécution des agents DIET. *TUNe* est système d'administration qui propose un niveau d'abstraction plus élevé pour décrire l'encapsulation des logiciels dans des composants, le déploiement de l'architecture et les politiques de reconfiguration à appliquer automatiquement. A la fin de ce chapitre, nous étudions les problèmes du passage à l'échelle dans le système TUNe.

3.1.1 DIET (Distributed Interactive Engineering Toolbox)

DIET [CLQS02] [CD06] est un ordonnanceur développé au sein du projet INRIA GRAAL, conjointement au LIP (à l'ENS Lyon) et au LIFC (à l'Université de Franche-Comté). Il permet de déployer un ensemble de serveurs de calculs sur des

clusters et d'en offrir l'accès à des clients répartis sur le réseau. La localisation de ressources et l'équilibrage de charges entre les serveurs se font grâce à un ensemble d'ordonnanceurs appelés agent eux-mêmes repartis sur le réseau. Les agents sont de trois types et organisés de façon arborescente : **MA** (Master Agent), **LA** (Local Agent) et **SeD** (Server Daemon). La figure 3.1 montre l'architecture arborescente de DIET.

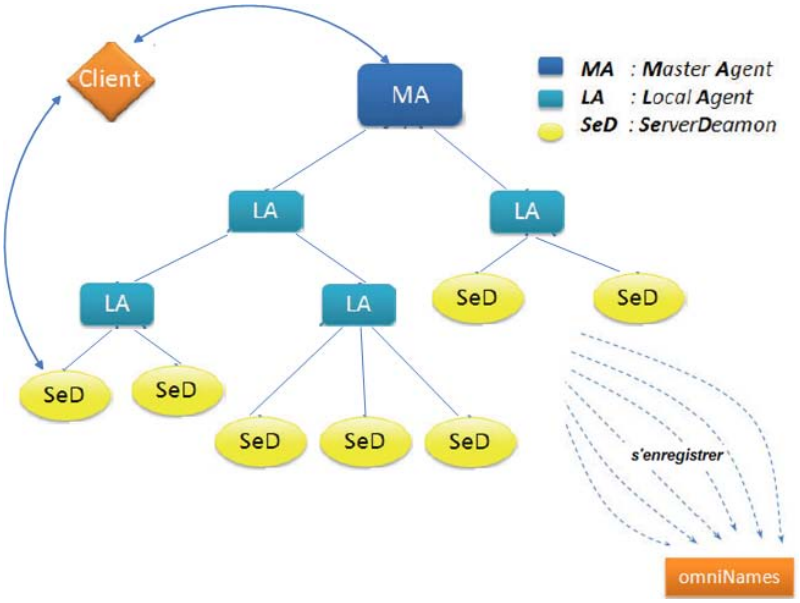

FIGURE 3.1 – Architecture de Diet

L'agent **MA** est directement relié aux clients. Il reçoit des requêtes de calculs des clients et choisit un ou plusieurs SeD ayant la capacité de résoudre le problème. Il a une vue globale sur toute l'architecture déployée.

L'agent **LA** constitue un niveau intermédiaire dans l'arborescence DIET. Il peut être le lien entre un MA et un SeD, entre un autre LA et un SeD ou entre deux LA. Son but est de diffuser les requêtes et les informations entre le MA et les SeD. Il tient à jour une liste des requêtes en cours de traitement et, pour chacun de ses sous-arbres, le nombre de serveurs pouvant résoudre un problème donné, ainsi que des informations liées aux données.

Un **SeD** est le point d'entrée d'un serveur de calculs. Il gère l'ensemble des ressources qui lui sont allouées. Il peut aussi s'agir d'un processeur que d'un cluster. Il tient à jour une liste des données disponibles sur un serveur, une liste des problèmes qui peuvent y être résolus, et toutes les informations concernant sa charge (charge CPU, mémoire disponible, taille du disque, etc.). Les SeD sont essentiellement chargés d'effectuer une sous-partie du calcul global en vue de la résolution du problème soumis par un client.

Un *client* est une application qui utilise DIET pour résoudre des problèmes. Différents types de clients sont en mesure de se connecter à DIET à travers un MA

depuis une page web, un environnement de résolution de problème tel que *Matlab* ou directement depuis un programme écrit en *C* ou en *Fortran*.

Pour soumettre un calcul, un client de DIET doit se connecter au *Master Agent* le plus approprié ou plus proche. Une fois que son *Master Agent* est identifié, le client peut lui soumettre un problème. Pour choisir le serveur le plus approprié pour résoudre ce problème, le *Master Agent* propage une requête dans ses sous-arbres afin de trouver à la fois les données impliquées et les serveurs capables d'effectuer l'opération demandée (produit matriciel, somme matricielle, etc.). Ensuite, le *Master Agent* renvoie l'adresse du serveur choisi au client et effectue le transfert des données persistantes impliquées dans le calcul. Le client communique ses données locales au serveur et alors la résolution du calcul peut être effectuée. Les résultats pourront être renvoyés au client en fonction du problème.

Le déploiement d'une application DIET nécessite un service de nommage CORBA (*omniNames*). Ce service doit être impérativement démarré avant les autres agents DIET (MA, LA, SeD), afin que ceux-ci puissent s'enregistrer auprès de lui sous une référence. Cet enregistrement permet par exemple à un client de retrouver un MA et à un LA de retrouver son père dans l'arborescence (qui peut être un MA ou LA). Après le démarrage du service de nommage, l'agent MA peut être démarré. L'agent localise le service de nommage et s'enregistre. Par la même procédure, les LA puis les SeD sont démarrés dans l'ordre approprié. Chaque agent s'enregistre dans l'annuaire du service de nommage et utilise ce dernier pour localiser son père dans l'arborescence.

3.1.2 LogService

LogService [gra0xa] est un système de surveillance qui relaye les messages et les informations qui surviennent dans une plate-forme distribuée. Il s'agit d'un système générique qui doit être interfacé avec l'application à surveiller. Pour cela chaque élément à surveiller se voit attacher un gestionnaire spécifique (*LogComponent*), qui relate les évènements à un autre gestionnaire centralisé (*LogCentral*). *LogCentral* stocke l'information ou la transmet à des outils qui auront la charge de la traiter : les *LogTools*. Dans le cadre de la plate forme DIET, le LogService peut être utilisé pour détecter une sortie anormale des agents de DIET. Lorsqu'il détecte la panne d'un agent, il marque son état défectueux comme étant confus et attend une éventuelle intervention de l'administrateur ou un programme approprié pour entreprendre les diagnostics de la panne.

3.2 Contexte du projet

Système d'administration autonome *TUNe* : **T**oulouse **U**niversity **Ne**twork

TUNe [BSB+08], [BHS+08] est un système autonome fondé sur un modèle à composants. Il a été dévéloppé dans la thèse de L.Broto [Bro08] et offre une vision uniforme d'un environnement logiciel composé de différents types de logiciels. Chaque logiciel administré est encapsulé dans un composant et l'environnement logiciel est abstrait sous la forme d'une architecture à composants. Le modèle à composants utilisé pour l'implantation de TUNe est Fractal. La section suivante présente les principales caractéristiques de ce modèle.

Le modèle à composants Fractal

Le modèle à composants Fractal est basé sur les notions de *composants*, *interfaces* (avec *interfaces de contrôle*) et *liaisons* [BCL+06b]. Un *composant* est une entité exécutable conforme au modèle Fractal. Ce modèle distingue deux types de composants : les composants primitifs et les composants composites. Les composants primitifs encapsulent une unité de calcul décrite dans un langage de programmation. Les composants composites encapsulent un groupe de composants primitifs et/ou d'autres composites. Une *interface* est un point d'accès au composant. Une interface implante un type d'interface qui spécifie les opérations supportées par cette dernière. Il existe deux catégories d'interfaces :

- les *interfaces serveur* qui sont des points d'accès acceptant des appels de méthodes. Elles correspondent donc aux services fournis par le composant ;
- les *interfaces client* qui sont des points d'accès émettant des appels de méthodes. Elles correspondent aux services requis par le composant.

La *partie de contrôle* expose les interfaces du composant et comporte des objets contrôleurs et intercepteurs. Le modèle Fractal ne contraint pas la nature des contrôleurs contenus dans la partie de contrôle. Il est ainsi possible d'exercer un contrôle adapté sur les composants. La librairie Fractal fournit quatre contrôleurs :

- Le *contrôleur d'attributs* permet de modifier les attributs primitifs d'un composant. Ces attributs incluent les types primitifs (booléens, entiers, etc.) et les chaînes de caractères ;
- Le *contrôleur de liaisons* permet d'établir ou de rompre une liaison primitive entre les interfaces client du composant qui possède le contrôleur, et une ou plusieurs interfaces serveurs d'autres composants ;
- Le *contrôleur de contenu* pour les composites permet d'ajouter et de retrancher des sous-composants à un composant composite ;
- Le *contrôleur de cycle de vie* permet de contrôler le cycle de vie du composant qui possède le contrôleur. Le cycle de vie d'un composant est représenté par un automate à deux états : *started* (le composant est dans un état démarré) et *stopped* (le composant est dans un état d'arrêt). Le contrôleur permet de passer d'un état à l'autre.

Une *liaison* est un canal de communication établi entre les composants Fractal. Cette communication est uniquement possible si les interfaces (client/serveur) de

ces composants sont liées. La figure 3.2 décrit les différentes entités mises en jeu dans une architecture de type Fractal. Le rectangle noir représente le contrôle du composant alors que l'intérieur du rectangle représente le contenu du composant. Les flèches correspondent aux liaisons tandis que les formes en T attachées aux rectangles correspondent aux interfaces du composant. Les interfaces de contrôle sont représentées par les lettres telles que le contrôleur de composant (c), le contrôleur de cycle de vie (lc), le contrôleur de liaisons (bc), le contrôleur de contenu pour le composant composite (cc) ou le contrôleur d'attributs (ac).

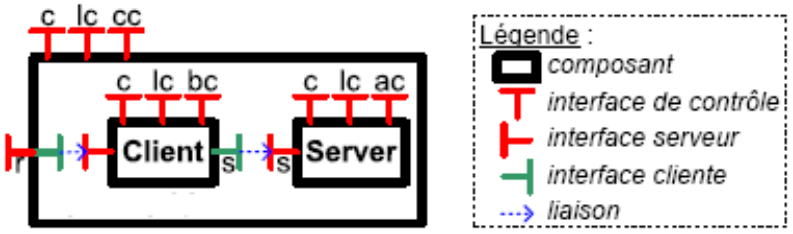

FIGURE 3.2 – Composant client/serveur

L'application décrite dans la figure 3.2 se compose de deux composants primitifs : **Client** et **Server**. Ces composants sont liés contractuellement par une interface *Service* nommée *s*. Celle-ci est une interface cliente pour le composant **Client** et elle est serveur pour le composant **Server**.

Tout logiciel géré par TUNe est encapsulé dans un composant Fractal qui fournit une interface permettant son administration. Ainsi, le modèle à composants est utilisé pour implanter une couche d'administration qu'on appelle SR (System Representation)(Figure 3.3) au dessus de la couche patrimoniale (composée des logiciels administrés).

La couche SR contient la représentation à composants de l'application et de l'infrastructure matérielle. Ces composants sont basés sur le modèle Fractal et fournissent une interface d'administration pour les logiciels encapsulés dont le comportement est spécifique au logiciel. Cette interface permet ainsi de contrôler l'état du composant de manière homogène en évitant des interfaces de configuration complexes et propriétaires. **La couche patrimoniale**, est la couche où s'exécute l'application patrimoniale. Elle est répartie sur plusieurs nœuds sous forme de démon. Lors du déploiement de l'application patrimoniale, TUNe va en première approximation déposer sur chaque nœud un morceau de l'application, le configurer puis le démarrer suivant un ordre décrit par l'administrateur. Pour cela, l'administrateur dispose d'une interface lui permettant de décrire les différents éléments et le processus d'administration. Une vue générale de cette interface est illustrée sur la Figure 3.4.

– **Profils UML** : L'approche TUNe introduit un profil UML pour décrire graphiquement le déploiement. L'un des avantages est qu'UML est plus intuitif

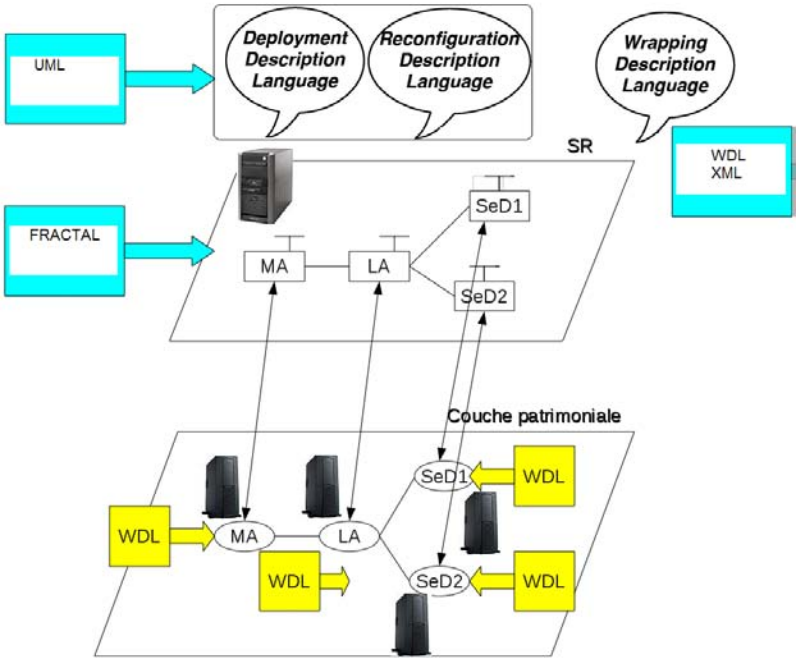

FIGURE 3.3 – Couche d'administration dans TUNe

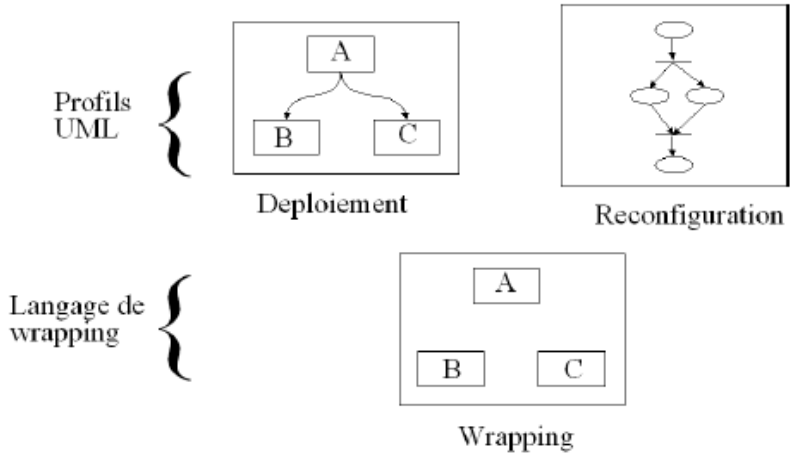

FIGURE 3.4 – Interface d'administration de TUNe

que le langage de description d'architecture basé sur des formalismes tels que XML.

– **Langage de wrapping** : Un Langage de Description de Wrapper WDL (Wrapper Description Language) permet de décrire le comportement des wrappers d'un logiciel. Ce langage permet de décrire les fonctions d'administration telles que la *configuration*, le *démarrage* et l'*arrêt* d'un logiciel. Ces différentes fonctions sont exécutées par TUNe lors de l'administration du logiciel.
– **Reconfiguration** : Un autre profil UML est introduit pour spécifier les règles de reconfiguration grâce à un diagramme d'état-transition. Ces diagrammes

sont utilisés pour définir le *workflow* des opérations qui doivent être exécutées pour reconfigurer l'environnement administré.

Nous présentons dans la suite de cette section l'utilisation générale et le fonctionnement du système TUNe. Nous décrivons en détail les différents diagrammes et le langage d'encapsulation des logiciels ainsi que les étapes à suivre pour déployer une application avec TUNe.

1. **Description de l'infrastructure matérielle**
   Dans le cas de l'administration avec TUNe, les applications sont décrites en utilisant les profils UML. A cet effet un profil UML est introduit afin de spécifier le diagramme de description de la grille. Ce diagramme a pour rôle de donner une description de l'infrastructure matérielle. Une grille est représentée par un ensemble de clusters. Chaque cluster est représenté par une classe UML. Les caractéristiques d'un cluster sont définies sous forme d'attributs et sont communes aux nœuds du cluster. Un exemple de caractéristique peut être : le chemin d'installation de java, protocole de connexion à distance utilisable (ssh, ftp...) etc. La figure 3.5 montre la description d'une infrastructure matérielle composée de trois clusters (**toulouse**, **bordeaux** et **lyon**). Dans un souci de lisibilité, certaines caractéristiques ne sont mentionnées sur la figure. Parmi les caractéristiques primordiales d'un cluster, on peut citer :
   - *user* [OPTIONNEL]. Cet attribut contient le login de l'utilisateur pour se connecter au nœud distant. S'il n'est pas défini, l'utilisateur courant est utilisé ;
   - *dirlocal* [OBLIGATOIRE]. Cet attribut contient le nom du répertoire où le déploiement s'effectuera sur le nœud distant. Il correspond au répertoire d'installation des paquetages et les librairies du logiciel ;
   - *javahome* [OBLIGATOIRE]. Cet attribut contient l'emplacement sur le nœud distant de la machine virtuelle java nécessaire à l'exécution des démons de TUNe ;
   - *protocole* [OPTIONNEL]. Cet attribut indique le protocole de connexions sur les nœuds d'un cluster. Il peut prendre comme valeur *ssh* ou *oarsh* (protocole de connexion utilisé sur Grid 5000, proche de *ssh*) et sera le protocole de copie du logiciel patrimonial et d'exécution à distance des démons sur le nœud distant. S'il n'est pas défini, le protocole *ssh* est choisi par défaut ;
   - *allocator* [OBLIGATOIRE]. Cet attribut donne le nom d'un fichier qui va contenir un ensemble de politiques d'allocation qu'on peut utiliser pour allouer les nœuds du cluster aux entités logicielles ;
   - *allocator-policy* [OBLIGATOIRE]. Cet attribut donne le nom de la politique d'allocation courante à utiliser parmi celle décrites dans le fichier défini par l'attribut *allocator* ;
   - *nodefile* [OBLIGATOIRE]. Cet attribut représente le nom d'un fichier qui contient les noms des nœuds du cluster.
2. **Description du diagramme de configuration**
   Pour déployer une application avec TUNe, il faut décrire son architecture en utilisant le formalisme UML. Un profil UML qu'on appelle dans la suite *dia-*

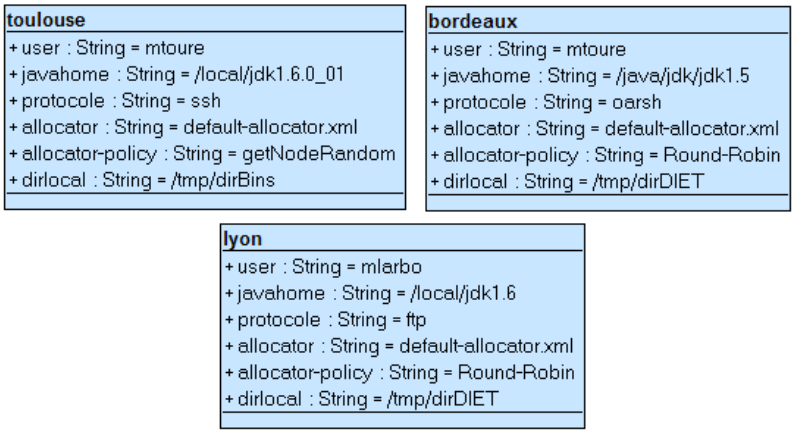

FIGURE 3.5 – Diagramme de description de l'infrastructure matérielle

gramme de configuration est introduit pour configurer chaque entité logicielle et l'architecture logicielle de l'application. Lors du déploiement, ce diagramme est interprété pour déployer une architecture à composants. Chaque élément (boîte) correspond à une entité logicielle et les associations entre les éléments correspondent aux dépendances entre les entités logicielles. Pour une application DIET, la configuration des trois entités MA, LA, SeD est représentée à la figure 3.6. Dans ce diagramme, chaque élément représente un agent DIET. Ce diagramme permet de déployer une application DIET composée d'1 MA, d'1 LA et 2 SeD.

Des attributs sont définis dans chaque élément. Une partie de ces attributs est prédéfinie et l'utilisateur doit en donner les valeurs, une autre partie est aussi prédéfinie mais TUNe affecte les valeurs et enfin une partie est à la discrétion de l'utilisateur lui permettant de décrire les paramètres de configuration d'une entité logicielle. Les attributs prédéfinis à remplir par l'utilisateur sont les suivants :

- *wrapper* [OBLIGATOIRE]. Cet attribut doit avoir comme valeur par défaut le nom du fichier wrapper qui définit les actions applicables à ce type de composant (voir la section suivante pour plus de détails) ;
- *software* [OPTIONNEL]. Cet attribut doit avoir comme valeur par défaut le nom du fichier tgz (TAR Gunzip, un format d'archivage compressé) qui contient l'application patrimoniale à déployer pour ce type de composant ;
- *host-family* [OBLIGATOIRE]. Cet attribut permet de déclarer sur quelle famille de nœud les composants de ce type doivent être déployés. Ces noms correspondent aux noms des clusters du diagramme de nœuds.

Les attributs prédéfinis et remplis par TUNe sont les suivants :

- *nodeName*. Cet attribut représente le nom du nœud sur lequel le logiciel patrimonial a été déployé. Le nœud est alloué par un allocateur de nœuds ;
- *tubeAddr*. Cet attribut indique le nom d'un tube de communication inter-processus qui permet une instance logicielle notamment les sondes, d'envoyer des notifications à TUNe ;

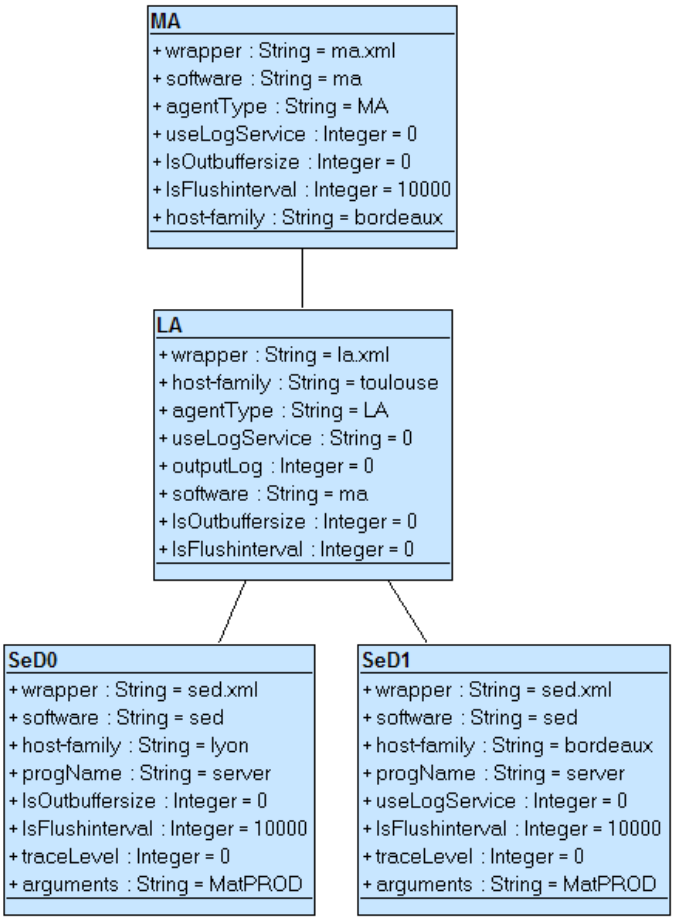

FIGURE 3.6 – Exemple d'un profil UML pour une architecture DIET

– *srname*. Cet attribut désigne le nom unique du composant au sein du SR. Ce attribut peut être utilisé lors de la reconfiguration pour identifier de façon unique une entité logicielle.

3. **Définition des wrappers**

Comme évoqué précédemment, chaque entité logicielle du diagramme de configuration possède un wrapper spécifié par l'attribut *wrapper*. Ce wrapper représente l'encapsulation de l'entité permettant ainsi d'interagir avec elle (les opérations de configuration, reconfiguration, démarrage etc.). L'encapsulation consiste à définir l'interface de contrôle de l'application. Cette interface de contrôle définit les méthodes pouvant être appelées depuis les diagrammes de (re)configuration. Un wrapper va donc contenir : les méthodes avec les paramètres d'appel, les classes java qui implantent ces méthodes. Il faut distinguer deux parties : une partie spécification décrite dans un langage WDL (**Wrapping Description Language**) et une partie code java qui implante les méthodes correspondantes aux fonctions d'administration d'une entité logicielle.

La partie spécification du langage WDL permet de décrire les fonctions d'administration sous forme de méthode avec des paramètres passés à ces méthodes.

C'est un langage qui possède une syntaxe XML et qui utilise seulement trois balises :

- *wrapper* avec l'attribut *name* qui définit le nom du wrapper ;
- *method* avec les attributs :
  - *name* qui sera le nom de la méthode donné dans les états d'action des diagrammes de reconfiguration ;
  - *key* qui est le nom de la classe qui inclut le code la méthode ;
  - *method* qui est le nom de la méthode java dans la classe *key*.
- *param* qui sont les paramètres à passer à la méthode.

L'intérêt du WDL est d'utiliser la notation pointée dans la balise *param* en vue de naviguer dans le SR et passer des arguments aux méthodes java. Ainsi, dans la balise *param*, une chaîne

- qui commence par $ signifie qu'un attribut doit être récupéré dans le SR. Tous les attributs prédéfinis ou définis par l'utilisateur peuvent être utilisés. Cette chaîne aura comme composant source le composant sur lequel la méthode doit s'appliquer ;
- qui ne commence pas par $ sera passée telle quelle à la méthode java.

Pour naviguer dans le SR, on part du composant courant comme source de la navigation. Ainsi, $*C1.attribut* ira chercher l'attribut *attribut* dans le composant *C1* relié au composant courant. Si plusieurs composants *C1* sont reliés, les attributs seront récupérés et séparés par des ";". Pour une architecture DIET, *LA.MA.nodeName* ira chercher à partir d'un composant SeD, le nom de la machine (attribut *nodeName*) sur laquelle s'exécute l'entité logicielle MA qui est reliée au LA qui est à son tour relié au SeD courant. Cette navigation est effectuée en suivant les liaisons Fractal entre les composants wrappers des instances logicielles du SR.

Le mot clé *node* est ajouté à ce langage WDL pour pouvoir accéder au nœud sur lequel le composant est déployé. La figure 3.7 illustre nos propos.

Le fichier WDL présenté figure 3.8 définit 4 méthodes pouvant être appelées depuis les diagrammes de reconfiguration : *start*, *stop*, *configureOmni* et *configure*.

La figure 3.8 montre un exemple de spécification WDL qui encapsule un serveur de calcul DIET (SeD). On retrouve dans cet exemple la définition des méthodes *start* et *stop* qui sont appelées pour lancer ou arrêter le SeD. Une méthode de configuration est aussi définie permettant de refléter l'état du composant SeD (ses attributs de configuration) dans son fichier de configuration. Une définition de méthode inclut la description des paramètres à passer à la méthode quand cette méthode est appelée. Ces paramètres peuvent être des chaînes de caractères constantes, des valeurs d'attributs ou une combinaison des deux. Tous les attributs définis dans le diagramme de configuration peuvent être utilisés comme paramètres de méthode.

Intéressons nous à l'action *start*. Elle est définie dans la classe *extension.GenericStart* et se nomme *start_cmd* dans cette classe. Le premier paramètre correspond à la ligne de commande qu'il faut exécuter pour démarrer le serveur. Les autres paramètres sont passés comme deuxième et troisième paramètre à la méthode et sont des variables d'environnement nécessaires à l'exécution du serveur.

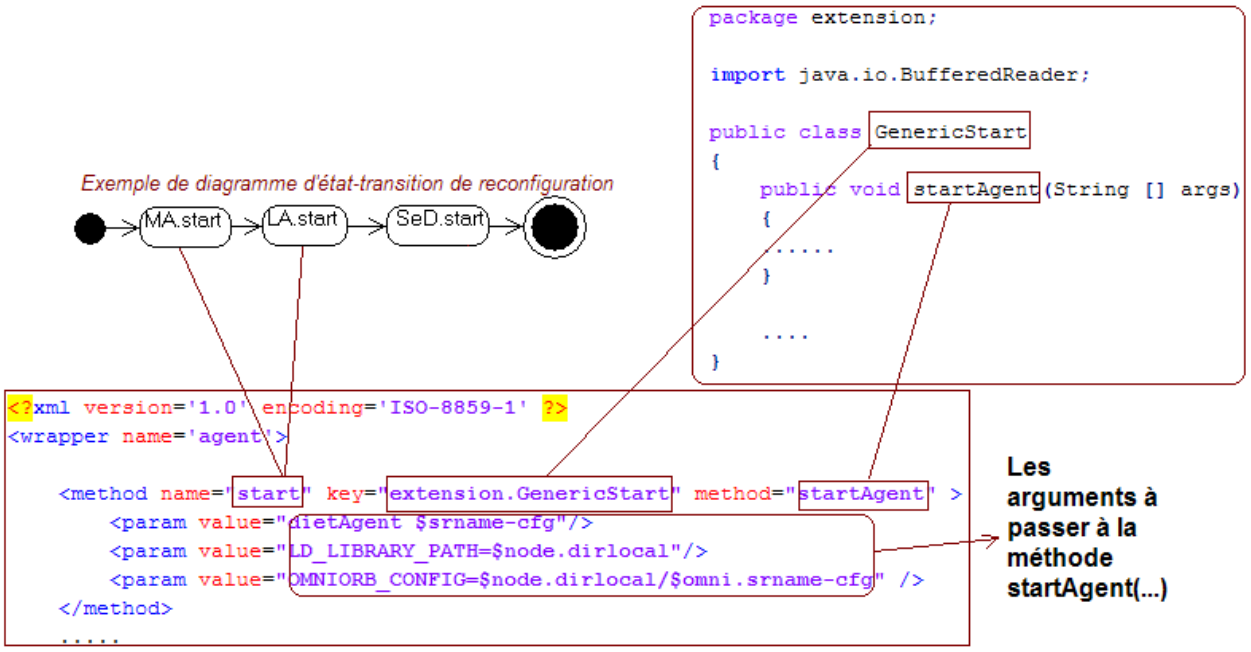

FIGURE 3.7 – Spécification d'un fichier WDL

4. **Définition du diagramme de démarrage et de reconfiguration de l'application**

   Un diagramme état-transition d'UML spécifique nommé *startchart* est introduit afin de donner l'ordre de démarrage des entités logicielles de l'application. Il permet de spécifier une suite d'actions à effectuer, l'ordre de ces actions et le niveau de parallélisations souhaité entre elles. Des états particuliers sont utilisés pour paralléliser, attendre, débuter et finir ces actions. Un état du diagramme est composé d'une entité définie dans le diagramme de configuration et l'action à effectuer sur cette dernière. Chaque action correspond à une méthode définie dans le wrapper de l'entité concernée. L'action *SeD.start* applique l'action *start* sur les deux entités logicielles SeD. Cette action est définie dans le wrapper de l'entité logicielle SeD. Un exemple du diagramme de démarrage est présenté sur la figure 3.9.

   Le diagramme de démarrage de la figure 3.9 permet tout d'abord de configurer de façon parallèle les entités logicielles de l'application en appliquant la méthode *configure* sur chacune d'elle (MA, LA, SeD0, SeD1). Après cette configuration, l'application est démarrée en utilisant la méthode *start* de chaque entité logicielle (MA, LA, les 2 SeD en parallèle).

   Dans un environnement autonome, des sondes sont conçues pour surveiller les applications et émettre des notifications. Un système d'administration autonome doit entreprendre l'exécution des opérations définies par l'administrateur afin de réagir à ces notifications. Dans le cas du système TUNe, les sondes ou d'autres applications peuvent émettre des notifications grâce à un tube de communication(tel que les pipes UNIX). En effet chaque fois qu'un composant wrapper d'une entité logicielle est instancié au niveau SR, un tube de

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<wrapper name='sed'>
    <method name="start" key="appli.wrapper.util.GenericStart"
                         method="start_with_pid_linux" >
        <param value="$dirLocal/$progName $dirLocal/$srname-cfg $arguments"/>
        <param value="LD_LIBRARY_PATH=$dirLocal"/>
    </method>

    <method name="configure" key="appli.wrapper.util.ConfigurePlainText"
                             method="configure">
        <param value="$dirLocal/$srname-cfg"/>
        <param value=" = "/>
        <param value="traceLevel:$traceLevel" />
        <param value="parentName:$LA.srname"/>
        <param value="name:$srname"/>
        <param value="lsOutbuffersize:$lsOutbuffersize"/>
        <param value="lsFlushinterval:$lsFlushinterval"/>
    </method>

    <method name="configureOmni" key="extension.GenericConfigurePlainText" method="conf
<param value="$dirLocal/$omni.srname-cfg"/>
<param value=" = "/>
<param value="InitRef:NameService=corbaname::$omni.nodeName:$omni.port" />
<param value="DefaultInitRef:corbaloc::$omni.nodeName" />
</method>

    <method name="stop" key="appli.wrapper.util.GenericStop"
                        method="stop_with_pid_linux" >
        <param value="$PID"/>
    </method>
</wrapper>
```

FIGURE 3.8 – Wrapper d'un SeD

communication est créé au niveau patrimonial permettant à cette entité logicielle d'envoyer des notification à TUNe à travers ce tube. Ces notifications possèdent une syntaxe particulière permettant le passage de paramètre. On notera que l'utilisation de tubes permet à n'importe quel logiciel écrit dans n'importe quel langage de générer des notifications.

Une notification envoyée dans le tube associé au composant wrapper est transmise au nœud d'administration de TUNe (où s'exécute le SR) où un programme de reconfiguration peut alors être exécuté. Une notification est définie par un type, le nom du composant qui l'a générée et un argument (tous de type chaîne de caractère). Pour définir les réactions aux évènements, nous avons introduit un profil UML qui permet de spécifier les reconfigurations comme des diagrammes d'état. Ces diagrammes définissent les opérations (et leur enchainement) qui doivent être appliquées en réaction à une notification. Considérons une sonde qui surveille régulièrement un agent LA (si le *PID* existe sur le

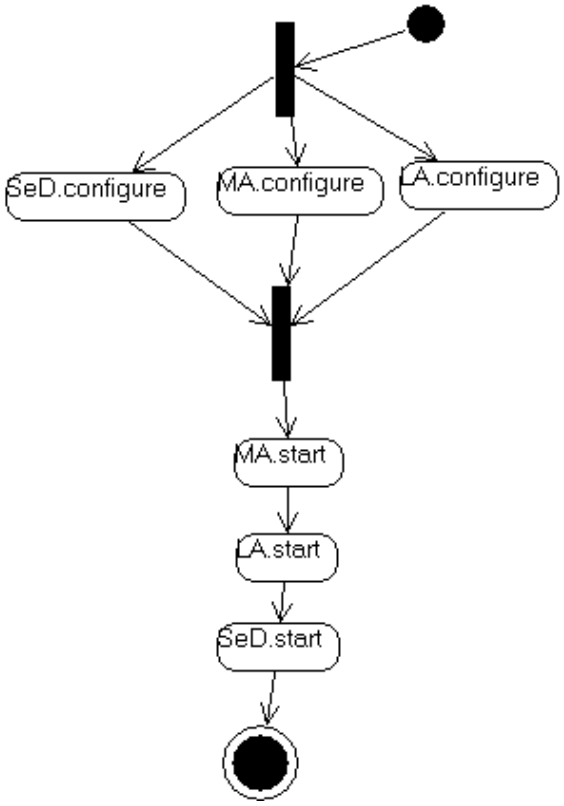

FIGURE 3.9 – Diagramme d'état-transition de démarrage d'une architecture DIET

nœud). Le diagramme de la figure 3.10, qui est la réaction à la notification correspondant à l'arrêt inopiné d'un LA dans Diet. Lorsque le *PID* du LA disparaît avec par exemple la commande kill ; la sonde notifie cette disparition en demandant à TUNe d'exécuter un diagramme de reconfiguration. Cette demande est effectuée par la sonde en écrivant dans le tube : le nom du diagramme à exécuter (*fixLA*), la variable *this* qui indique la sonde elle-même et le nom de l'instance *LA* à redémarrer indiqué par la variable *arg*.

Ainsi :

- *this.stop* appelle la méthode *stop* de la sonde pour éviter la génération d'autres notifications ;
- *arg.start* appelle la méthode *start* sur le LA pour le redémarrer. C'est la réparation proprement dite du LA ;
- *arg.SeD.stop* appelle la méthode *stop* des SeDs reliés au LA fautif. Ceci est nécessaire parce que dans Diet, le redémarrage d'un LA nécessite le redémarrage de tous ses SeDs fils pour qu'ils se réinitialisent et se reconnectent à leur LA père ;
- *arg.SeD.start* appelle la méthode *start* des SeDs reliés au LA fautif afin de redémarrer tous les SeD.
- *this.start* redémarre la sonde du LA.

5. **Déploiement de l'application**

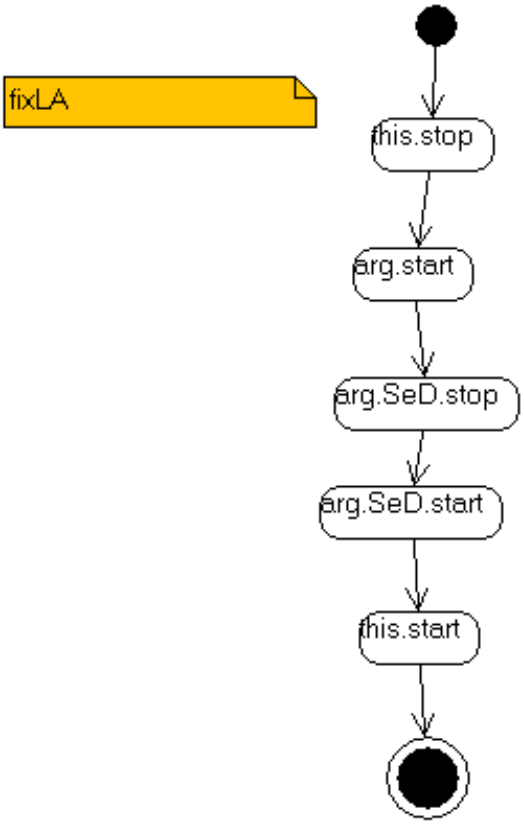

FIGURE 3.10 – Diagramme de réparation d'un LA

Après avoir décrit les wrappers des entités logicielles ainsi que le diagramme de configuration et le diagramme d'état-transition permettant de démarrer l'application, le déploiement proprement dit peut débuter. Pour cela, TUNe est exécuté sur un nœud central. Ce nœud peut être situé sur un cluster quelconque de la grille. Ainsi toutes les tâches d'administration sont effectuées sur ce dernier. A partir de ce nœud, TUNe commence par faire une projections entre le diagramme de déploiement et celui qui décrit la grille. Chaque entité est déployée sur son *host-family*. Ainsi l'entité MA et SeD1 sont déployées sur le cluster *bordeaux*, l'entité LA sur *toulouse* et SeD0 sur le cluster *lyon*. La figure 3.11 montre la projections effectuée par TUNe lors du déploiement.

6. **Lancement de TUNe**
   TUNe est lancé avec comme paramètre le fichier UML contenant les différents diagrammes (diagramme de configuration, de nœuds, de reconfiguration) au format XML généré. Lors du lancement de l'application, le fichier UML est parsé grâce à un parseur XML SAX (Simple API for XML). Ce parseur permet d'extraire les diagrammes pour générer l'architecture à composants dans le cas du diagramme de configuration et de nœuds. Dans la suite de cette section, nous décrivons le déroulement de la génération du SR (*Systèm Representation*). Dans un premier temps, nous présentons la génération de l'architecture à composants de la partie application puis nous terminons par la partie concernant la représentation à composants de la grille.

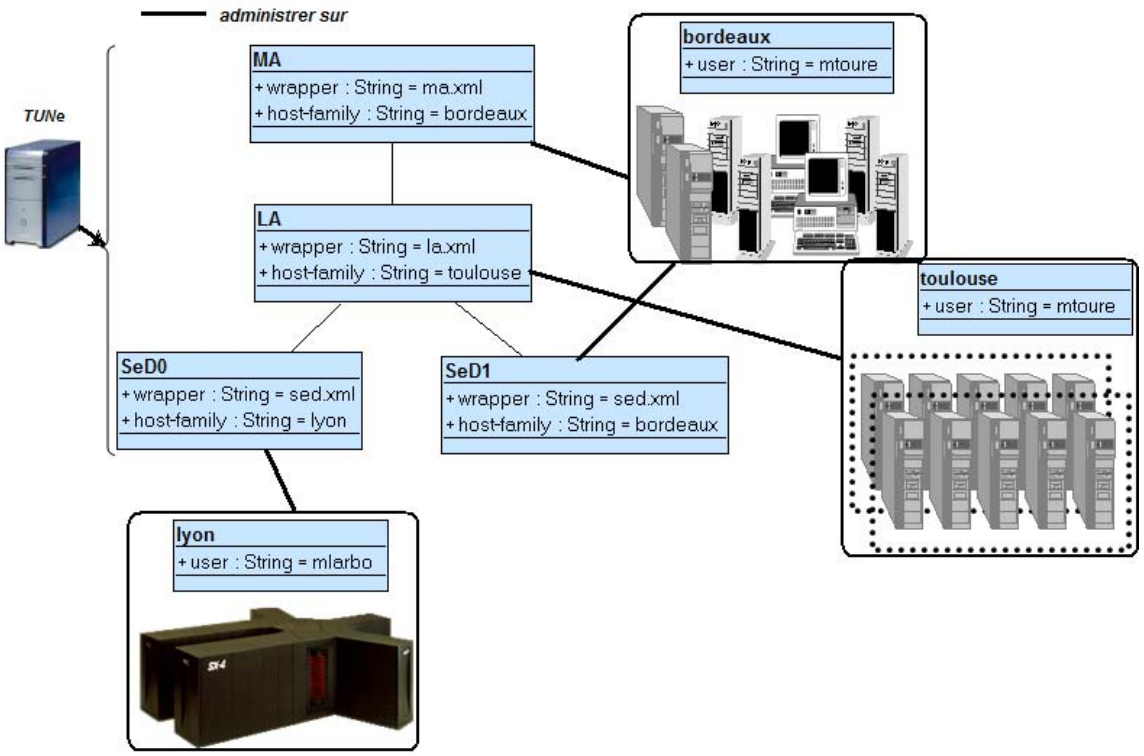

FIGURE 3.11 – Projection entre le diagramme de déploiement et l'infrastructure matérielle

La partie application : Lorsque TUNe parse le fichier UML, il génère automatiquement l'architecture à composants de l'application. Pour le diagramme de configuration, il existe des règles de passage de l'architecture de l'application en formalisme UML vers l'architecture du modèle à composants Fractal :

- un élément correspond à un composant Fractal,
- une association entre deux éléments donnera une liaison entre les deux composants Fractal,
- un attribut d'élément correspond à un attribut de composant.

Chaque composant est créé avec un contrôleur d'attribut générique qui permet de récupérer n'importe quel attribut grâce à un *get* ou un *set*, un contrôleur de liaison et un contrôleur de cycle de vie. La figure 3.12 montre un exemple de génération de SR pour le diagramme de configuration d'une application DIET. Ce diagramme est composé d'un MA, un LA et deux SeD (SeD0, SeD1).

La partie nœud : Les règles de passage pour ce diagramme sont plus simples car chaque élément qui représente un cluster donne un seul composant Fractal et peut être vu comme un *allocateur de nœuds*. Chaque nœud du cluster est encapsulé dans un composant Fractal (*Node*). Lors de la génération du SR, chaque composant d'une entité logicielle est relié au composant cluster (donc à son allocateur de nœuds) pour pouvoir allouer un nœud. Le composant d'un nœud alloué à une entité logicielle est également relié au composant de cette entité logicielle lui permettant d'accéder aux caractéristiques de son nœud d'administration (le nœud sur lequel l'entité logicielle est administrée).

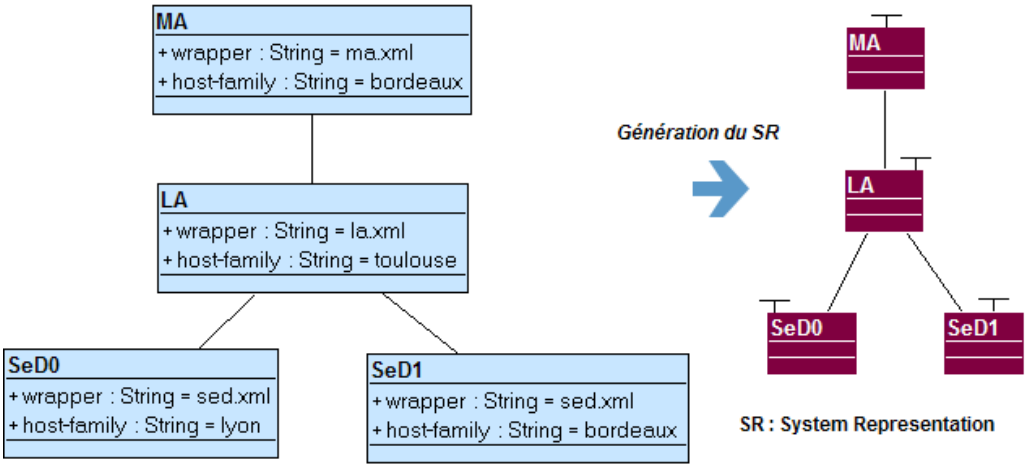

FIGURE 3.12 – génération du Systèm Representation (SR)

Synthèse :

Nous avons présenté dans cette section les applications **DIET** et **LogService** utilisées pour nos expérimentations, ainsi que le projet **TUNe** auquel nos contributions sont implantées.

DIET est un ensemble hiérarchique d'agents pour l'élaboration d'applications basées sur des serveurs de calculs qu'on appelle SeD. Les serveurs de calculs sont chargés d'exécuter des requêtes soumises par un client. Le client DIET, a pour fonction de soumettre les requêtes aux agents (MA, LA), puis de récupérer le serveur choisi par ces agents afin d'effectuer le calcul. Les agents ont la charge de localiser le ou les serveurs les plus adaptés à la requête (type de problème, capacité des ressources, disponibilité du serveur, etc.) et de retourner cette information au client DIET.

LogService est une application de génération de fichier de log pour un environnement distribué tel que DIET. Il a été utilisé dans cette thèse pour surveiller l'état des agents de DIET. En cas de défaillance d'un agent, le LogService est capable de la détecter.

TUNe est un système autonome basé sur un formalisme de haut niveau. Il propose un profil UML (*diagramme de configuration*) pour décrire l'architecture logicielle et les paramètres de configuration de l'application. Un autre profil UML est utilisé pour décrire les caractéristiques des différents clusters de la grille (*diagramme de nœud*). Chaque élément du diagramme de configuration est une entité logicielle avec comme attributs ses paramètres de configuration. L'association UML est créée entre les entités logicielles afin de décrire les dépendances entre les entités logicielles de l'application. Un élément du *diagramme de nœuds* représente un cluster avec comme attribut les caractéristiques matérielles ou logicielles des nœuds du cluster. Des diagrammes d'état-transition sont proposés pour décrire l'enchainement des opé-

rations à exécuter lors de la reconfiguration d'une application. Ces diagrammes sont également utilisés pour décrire le processus et l'ordre de démarrage d'une application (*startchart*). Ces opérations sont décrites dans un fichier WDL (*Wrapper Description Language*). Ce fichier WDL contient un langage dérivé de XML et qui permet de décrire les opérations d'administration telles que *configure*, *start*, *stop*, etc. pouvant être effectuées sur une entité logicielle décrite dans le diagramme de configuration.

Les problèmes du passage à l'échelle dans TUNe

Pour conclure cette section, nous présentons les problèmes du passage à l'échelle dans le système TUNe. Des expériences à grande échelle ont été menées avec le système TUNe en utilisant l'application DIET afin de montrer les problèmes du passage à l'échelle. Durant ces expériences, les problèmes soulignés sont :

- ***Administration centralisée*** : Toutes les tâches d'administration sont orchestrées et exécutées par TUNe à partir d'une machine centrale. Ainsi les tâches élémentaires du déploiement (création du répertoire d'installation, copie des ressources logicielles et éventuellement génération des fichiers de configuration) sont exécutées depuis le nœud d'administration. Dans un contexte de grande échelle, cela peut engendrer un problème de *performance* et constitue un goulot d'étranglement (la panne du nœud d'administration entraîne le dysfonctionnement total du processus d'administration). En effet l'administration d'une application peut demander beaucoup de ressources en terme de processus (pour l'exécution des tâches sur les machines distantes) et de connexions réseau. L'utilisation de ces ressources croît en fonction du nombre de nœuds sur lequel l'administration est effectuée et le nombre d'instances d'entités logicielles à administrer. Or le nombre de nœuds et d'entités logicielles est important dans un contexte de grande échelle et les ressources d'une machine sont limitées. Une seule machine ne peut donc pas se charger de l'exécution de toutes les tâches d'administration ;
- ***Installation*** : Le processus d'installation est implanté *en dur* dans le code source de TUNe. Le déploiement d'une application commence automatiquement par l'enchainement des processus de création de répertoire d'installation, copie des fichiers binaires et librairies, etc. L'administrateur n'a aucun contrôle sur l'exécution de ces processus. Le contrôle de la phase d'installation peut être utilisé par exemple pour augmenter la *performance* lors de l'administration en personnalisant les tâches d'installation (diminuer le nombre de copies, utiliser les paquetages existants sur les machines, etc.). Il peut également être utile pour tenir compte de l'environnement d'exécution notamment la topologie réseau (utilisation de NFS,...) ;
- ***Gestion de l'hétérogénéité*** : La gestion de *l'hétérogénéité* logicielle est effectuée par l'administrateur. Lors de la description de l'application, l'administrateur associe pour chaque entité logicielle, un paquetage sous forme *tgz* contenant les fichiers archives binaires et les librairies. Chaque entité logicielle possède également un attribut *host-family* qui indique la famille de machines sur laquelle elle est administrée. Ainsi le paquetage associé à l'entité logicielle lors de sa description doit être compatible aux machines de son *host-family*.

Cette contrainte doit être vérifiée par l'administrateur afin de s'assurer du bon déroulement de l'administration. Une grille de machines est décrite par TUNe sous forme d'un ensemble de classes de machines. Les classe sont indépendantes et possèdent chacune des caractéristiques représentées par des attributs. Ces caractéristiques sont communes aux machines de la classe. Cette description donne peu d'informations sur l'hétérogénéité de la grille et se limite à la description des clusters de la grille de façon indépendante. Par exemple, on ne peut pas exprimer la structure hiérarchique d'une grille, ni la notion de sous cluster (la composition entre les clusters).

Deuxième partie

Etat de l'art

Chapitre 4

Plates formes d'administration à grande échelle

Table des matières

Introduction

Le domaine de recherche de l'administration d'applications réparties est très vaste. Il existe de nombreux travaux et des domaines de recherche qui peuvent avoir plus ou moins de liens avec l'administration de façon générale (description, installation, configuration, reconfiguration, gestion d'hétérogénéité, etc.). L'objectif de ce chapitre est de présenter certains de ces travaux sur l'administration de logiciels en insistant plus particulièrement sur leurs liens avec le facteur d'échelle.

Le processus de déploiement d'une application s'étend de la description des paramètres de configuration et de l'architecture logicielle de l'application jusqu'à son exécution effective et sa terminaison. La description consiste à exprimer, dans un formalisme convenu, la configuration de chaque entité logicielle de l'application, leur assemblage, leurs dépendances vis-à-vis de librairies ainsi que leurs dépendances par rapport à d'autres entités logicielles. La gestion d'une application permet de la

modifier en cours d'exécution ou d'effectuer des opérations de maintenance pour satisfaire aux besoins de ses utilisateurs ou pour prendre en compte la modification de leur environnement d'exécution. Cette phase d'administration est effectuée par des administrateurs au travers d'une infrastructure de contrôle chargée d'exécuter les procédures d'administration (déploiement et reconfiguration). Ces procédures d'administration s'avèrent complexes dans un contexte de grande échelle comme la grille et posent des multiples problèmes. Ces différents problèmes sont généralement dus d'une part au nombre de machines de l'infrastructure matérielle et d'autre part au nombre d'entités logicielles qu'on administre. Dans ce qui suit, nous rappelons les différents problèmes liés à l'administration d'applications réparties dans un contexte de grande échelle :

- ***le facteur d'échelle***, tant en terme de distribution géographique (les sites d'une grille sont localisés d'un pays à un autre, d'un continent à un autre) qu'en terme du nombre de nœuds et d'entités logicielles (des centaines des milliers de machines et d'entités logicielles). Cette problématique engendre des problèmes de :
  - *expressivité* liés à la description des éléments qui interviennent dans le processus d'administration (matériels et logiciels, processus de déploiement, etc.). En effet l'une des problématiques du déploiement est la description de l'infrastructure matérielle, de l'architecture logicielle et les paramètres de configuration de l'application. Cette description est d'autant plus complexe que le nombre d'entités logicielles et de ressources matérielles est grand. L'objectif ici est de proposer un langage commode, intuitif et adapté, tenant compte du facteur d'échelle ;
  - *performance* liés à la charge et le coût du processus d'administration. L'administration d'une application répartie dans un contexte de grande échelle a un coût et demande beaucoup de ressources pour l'exécution des tâches du déploiement et de maintenance de l'application. En terme d'échelle, il s'agit d'une grille de machines constituée de plusieurs dizaines de milliers de nœuds qui peuvent être répartis dans plusieurs régions et pays. Il s'agit également d'un nombre important d'entités logicielles à administrer. Administrer ces innombrables ressources matérielles et logicielles à partir d'une machine centralisée paraît difficile et peut éventuellement influencer la performance du système d'administration (temps de déploiement, temps de réaction, etc.) ;
  - *hétérogénéité* des ressources matérielles (architectures et puissance des ordinateurs, topologies et performances des réseaux de communication, etc.) et logicielles (systèmes d'exploitation, bibliothèques installées, etc.) pose de problèmes lors de l'administration. La prise en compte de cette hétérogénéité de la grille est nécessaire pour le bon fonctionnement du processus d'administration. Le système d'administration doit pouvoir déployer une version de l'application adéquate et compatible selon les caractéristiques matérielles et logicielles de chaque machine.
- ***structure dynamique (la dynamicité)*** : l'une des caractéristiques de la grille est sa structure dynamique c'est-à-dire de nombreux ajouts ou retraits de nœuds. Ce comportement dynamique peut être dû à des défaillances de nœuds

ou à des ruptures de liens réseau ou simplement à des déconnexions volontaires (dans le cas par exemple où un administrateur souhaite effectuer une mise à jour du système d'exploitation). La défaillance d'un nœud entraîne la perte de toutes les entités logicielles s'exécutant sur ce dernier. La panne d'une entité logicielle ou la coupure du lien réseau peut entraîner le dysfonctionnement partiel ou total de l'application.

La première problématique de l'administration d'applications réparties à grande échelle que nous avons abordée est liée aux caractéristiques de la grille en l'occurrence le facteur d'échelle (*performance*, *hétérogénéité*, *expressivité*). L'une des approches pour remédier au problème de *performance* est de diminuer la charge du gestionnaire chargé de l'administration de l'application. En effet, le déploiement de multiples entités logicielles sur des milliers de machines a un coût et peut nécessiter beaucoup de charges et de ressources. En diminuant cette charge, la réactivité augmente tout en diminuant le temps de déploiement permettant ainsi d'augmenter la performance du système d'administration. *L'hétérogénéité* nécessite un gestionnaire de déploiement permettant de sélectionner pour un nœud donné la bonne version de l'application qui lui est compatible au niveau du système d'exploitation, d'architecture processeur, etc. Le problème *d'expressivité* est lié à la description de l'infrastructure matérielle et logicielle. Le système d'administration doit proposer un formalisme simple d'utilisation, intuitif et tenant compte du facteur d'échelle. La dernière problématique est liée au comportement dynamique de la grille. Il s'agit ici de déconnexion aléatoire de machine, rupture du lien réseau, des pannes matérielles et logicielles qui sont très fréquentes dans le contexte de grille. Cette problématique peut être résolue par l'approche basée sur l'administration autonome qui permet de détecter automatiquement ces différents types de défaillances et de proposer des réparations sans intervention humaine.

Dans ce chapitre d'état de l'art, nous mettons l'accent sur les outils d'administration. Ainsi, nous avons divisé les outils en deux catégories : la première catégorie concerne les *système de déploiement*. Nous étudions ces systèmes en insistant plus particulièrement sur la mise en œuvre du processus de déploiement. La seconde catégorie porte sur les systèmes d'*administration autonome*. Ces systèmes effectuent l'administration d'une application de façon autonome (donc sans intervention humaine).

Pour chaque système, nous présentons :

- ***Une description générale*** : Nous introduisons dans cette partie, une brève présentation de l'architecture et les composants principaux du système ;
- ***L'expressivité*** : Dans cette partie, nous présentons les formalismes mis en œuvre pour décrire l'infrastructure matérielle et logicielle, éventuellement le processus de déploiement ;
- ***Le déploiement*** : Pour étudier la performance d'un système, nous insistons sur la répartition de charges du déploiement pour les systèmes de déploiement et de l'administration (le déploiement et l'auto-reconfiguration) pour les sys-

tèmes autonomes. Ainsi, nous présentons pour chaque système, l'architecture et l'approche mise en œuvre pour effectuer le déploiement/l'administration (centralisée ou décentralisée) afin d'étudier la performance et le passage à l'échelle ;
- ***Gestion de l'hétérogénéité*** : Dans le contexte de grande échelle, l'environnement peut être composé de divers ressources matérielles et logicielle hétérogènes. Nous présentons les approches mises en œuvre pour remédier au problème d'hétérogénéité ;
- ***Tolérance aux pannes*** : Pour finir, nous étudions pour chaque système, les solutions mises en œuvre pour tolérer les pannes pendant le déploiement, pour les systèmes de déploiement, et pendant le déploiement et l'exécution de l'application, pour les systèmes d'administration autonome.

4.1 Systèmes de déploiement

Dans cette section, nous présentons plus en détail les outils faisant partie de la thématique du déploiement d'applications sur un environnement de type grille ou grappe de machines : **ADAGE** [LPP05] [Lac05], **ORYA** [MB04] [CLM05], **Software Dock** [HHW99], **GoDiet** [CCD06],**Taktuk** [MR03]. Le premier système **ADAGE** est un système de déploiement d'applications parallèles de type MPI. **ORYA** est un environnement ouvert basé sur les outils de déploiement existant permettant d'automatiser les différentes activités du déploiement. **Software Dock** [HHW99] est un outil conçu pour déployer des applications de type client/serveur. **SmartFrog** [GGL+03] [Sab06] est un canevas distribué, dédié au déploiement d'applications réparties sur des grappes de machines. **GoDiet** [CCD06] est une infrastructure dédiée au déploiement d'applications parallèles multiparamétriques basées sur la plateforme d'exécution distribuée DIET [CLQS02]. Enfin **Taktuk** [MR03] est un outil de déploiement hiérarchique dédié aux grilles de calcul.

4.1.1 ADAGE

Description générale

ADAGE (**A***utomatic* **D***eployment of* **A***pplications in a* **G***rid* **E***nvironment*) [LPP05] [LPP04b] [Lac05] est un système de déploiement générique pour applications parallèles et distribuées, développé par l'équipe PARIS de l'IRISA Rennes. Il permet d'automatiser le déploiement d'une architecture logicielle. Afin de déployer une application, **ADAGE** nécessite plusieurs informations : la description spécifique dans un formalisme propre à l'application, une description des ressources, une description des paramètres liés au déploiement appelée paramètres de contrôle. Les paramètres de contrôle permettent de déclarer des contraintes relatives au déploiement, telles que les latences et bandes-passantes requises et la politique de déploiement à utiliser. La description spécifique de l'application est traduite en une description générique

(**GADe** : **G**eneric **A**pplication **De**scription) par un programme : *convertisseur de description spécifique*. A partir de ces informations, **ADAGE** génère un plan de déploiement. Ce plan contient une mise en correspondance entre les demandes de l'application et la description des ressources. Les ressources sont automatiquement sélectionnées. Ensuite le déploiement est réalisé en transférant les fichiers requis, puis l'application est configurée et démarrée. L'architecture est présentée par la figure 4.1.

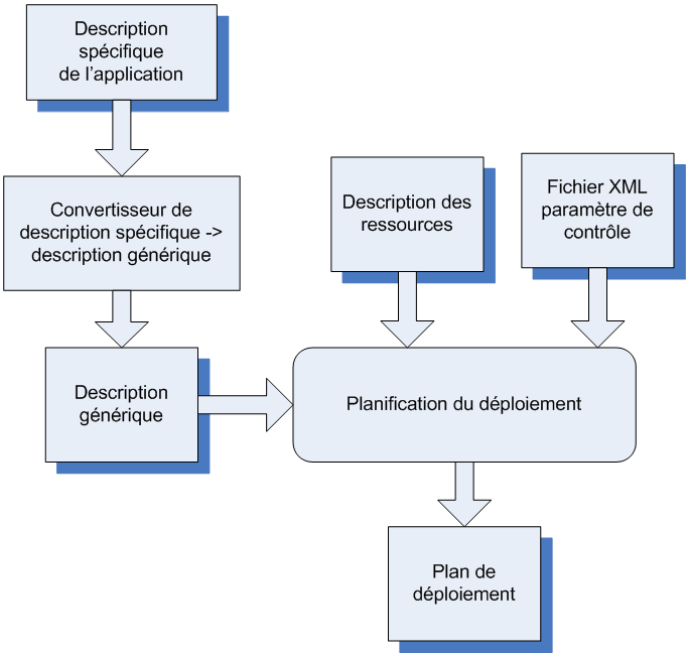

FIGURE 4.1 – Vue générale d'**ADAGE**

Expressivité

Pour expérimenter le déploiement avec le système **ADAGE**, les auteurs ont utilisé des applications du type MPI [WGR96] et GridCCM [PPR02]. À ce jour, il n'existe pas de formalisme standard pour décrire ces applications. A cet effet, **ADAGE** prend en compte l'existence de descriptions spécifiques à chaque type d'application qu'on veut déployer. Chaque description spécifique est rédigée dans un formalisme qui est propre au type de l'application. L'un des objectifs d'**ADAGE** est d'utiliser des formalismes existants de description spécifique d'une application. Cela permet aux administrateurs d'utiliser ces formalismes évitant ainsi l'apprentissage d'un nouveau formalisme. Cette description spécifique est écrite par les développeurs qui ont conçu l'application, et non par l'utilisateur ou l'administrateur qui veut la déployer. Elle (la description spécifique) doit être indépendante de tout environnement particulier afin de permettre facilement un déploiement dans d'autres environnements sans la modifier. Il s'agit ici par exemple d'une dépendance liée aux adresses IP, aux noms de machines. En plus de la description spécifique de l'application, **ADAGE** nécessite une description des ressources. Cette description définit l'ensemble des ressources pouvant être utilisées lors de la phase de planification. La description proposée par **ADAGE** permet d'exprimer de nombreuses

caractéristiques sur les ressources. Elles portent sur le matériel (processeurs, mémoire, disques, réseaux) et sur le logiciel (système d'exploitation, bibliothèques). Les caractéristiques des ressources matérielles sont décrites dans un format spécifique à **ADAGE**. Elles sont regroupées par appartenance à un même sous-réseau, ce qui permet de factoriser l'information et de passer à l'échelle. Cette description de ressources est effectuée en utilisant le formalisme XML et passée comme paramètre à **ADAGE** lors du déploiement d'une application. Pour pouvoir paramétrer le système de déploiement **ADAGE**, les auteurs proposent un fichier de description basé aussi sur le formalisme XML. Cette description appelée *paramètre de contrôle*, va contenir les paramètres de configuration d'**ADAGE**. Elle permet d'exprimer des contraintes fournies par l'utilisateur sur une exécution particulière de l'application. Ces contraintes peuvent donner des informations sur la qualité de service désirée pour ce déploiement, sur les caractéristiques des ressources pouvant être utilisées ou encore privilégier un algorithme de déploiement plutôt qu'un autre.

Déploiement

Le gestionnaire de déploiement pour **ADAGE** est entièrement centralisé. Toutes les tâches de déploiement sont effectuées à partir de la machine sur laquelle **ADAGE** est lancé. Le gestionnaire est chargé de l'exécution du processus de déploiement appelé *plan de déploiement*. Cette exécution comprend le transfert des fichiers vers les ressources cibles, le lancement des processus de l'application et la récupération d'un identifiant de tâche (pour surveiller ou terminer ultérieurement l'exécution d'une tâche). Après l'exécution du plan de déploiement, **ADAGE** génère un rapport de déploiement contenant notamment des informations sur le placement des processus et leurs identifiants.

Gestion de l'hétérogénéité

Le gestionnaire du déploiement est responsable de sélectionner les implémentations des programmes qui sont compatibles avec les systèmes d'exploitation et les architectures matérielles des ressources. Les besoins d'une application sont exprimés sous forme de contraintes. Lors de la description spécifique d'une application, des contraintes sont spécifiées notamment sur le système d'exploitation, sur l'architecture processeur compatible à l'application, etc. Ces contraintes peuvent être décrites dans le fichier de paramètre de contrôle. Le gestionnaire résout ces contraints lors de l'exécution du déploiement. Il tient compte des besoins exprimés par les applications.

Tolérance aux pannes

ADAGE donne la possibilité de surveiller une application en cours d'exécution. Cette surveillance peut être utilisée pour gérer les pannes logicielles de façon ad-hoc.

Synthèse

Le déploiement d'une application avec **ADAGE** nécessite un fichier XML qui contient ce qu'on appelle le *paramètre de contrôle*. Ce fichier décrit des informations indispensables à l'exécution du déploiement notamment les contraintes sur les machines qu'on peut utiliser, le login sur ces machines, les algorithmes à utiliser pour planifier le déploiement, etc. Une description spécifique de l'application est définie ainsi qu'une description des caractéristiques des ressources matérielles. **ADAGE** utilise une description générique de l'application définie au format XML générée à partir de sa description spécifique par un programme appelé *convertisseur de description spécifique*. Ces différentes descriptions sont utilisées pour produire un plan de déploiement. L'exécution de ce plan effectue le déploiement proprement dit.

ADAGE est un prototype qui présente quelques limites. Il ne dispose pas de l'expressivité nécessaire pour décrire les applications n'ayant pas de formalisme de description spécifique. Cette lacune limite sa généricité. La notion de convertisseur de descripteur limite l'utilisation du système. En effet, il faut un convertisseur de description spécifique pour chaque type d'application, ce qui présente de contraintes pénibles pour un administrateur. Enfin le déploiement est effectué de façon centralisée, ce qui limite le passage à l'échelle.

4.1.2 ORYA

Description générale

ORYA (**O***pen* envi**R***onment to deplo***Y** **A***pplications*)[MB04] [LB03] [CLM05] est un outil de déploiement d'applications pour les entreprises de taille moyenne. Il est développé au sein du laboratoire LISTIC et du LSR/ADELE. Il est basé sur l'utilisation d'outils de déploiement existants. L'objectif de ces travaux est de proposer un canevas permettant d'offrir un support qui automatise les différentes tâches de gestion du cycle de vie du déploiement. Le mécanisme de déploiement d'un logiciel consiste à sélectionner une entité logicielle, à l'installer et à la désinstaller. L'approche adoptée par **ORYA** est une approche dirigée par les modèles. Les concepts de déploiement sont donc décrits par un méta-modèle. Le méta-modèle d'**ORYA** propose de formalismes de description de l'environnement de l'entreprise, des machines cibles et les applications à déployer. Ce méta-modèle vise à représenter les concepts communs au déploiement et s'abstraire de toute notion spécifique. Enfin, un plan de déploiement est créé pour réaliser toutes les étapes du déploiement.

ORYA déploie des logiciels au sein d'une entreprise. Une vue générale de son architecture est représentée par la figure 4.2.

La figure 4.2 montre qu'**ORYA** est composé de trois serveurs : *serveur d'applications* qui produit l'application à déployer et la met à la disposition des clients (les entreprises) ; *Serveur d'entreprise* qui a la connaissance du réseau d'entreprise, il sait quelles versions de quelles applications sont installées sur chacune des machines ; *Ser-*

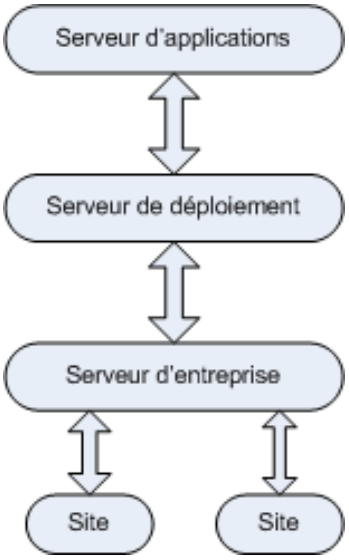

FIGURE 4.2 – Architecture de l'environnement **ORYA**

veur de déploiement gère le processus de déploiement. Des démons appelés *sites* sont installés sur les machines du réseau afin d'accéder aux caractéristiques matérielles et logicielles lors du déploiement.

Expressivité

ORYA propose un méta modèle [Bra0x] pour définir un format de description. Ce format permet de donner une description de l'infrastructure matérielle ainsi que les applications à déployer. Le méta modèle permet de définir trois types de modèle dans le cas du déploiement au sein d'une entreprise : *le modèle de l'entreprise*, le *modèle du site cible* et le *modèle du produit*. Le *modèle de l'entreprise* donne la description de la structure hiérarchique de l'entreprise composée de groupes d'entités et de sous groupes (organisation des machines par département, par fonction, etc.). Le *modèle du site cible* décrit les machines physiques nécessaires au déploiement avec les caractéristiques matérielles et logicielles. Les caractéristiques matérielles sont décrites par des propriétés comme la taille mémoire, l'espace disque, le processeur, etc. Les caractéristiques logicielles décrivent l'ensemble des unités déployées sur une machine. Les propriétés permettent de décrire les caractéristiques logicielles comme les paramètres de configuration. Enfin le *modèle du produit* décrit les différentes entités qu'on déploie avec les contraintes et les dépendances. La description de chaque machine est donnée dans un fichier XML représentant le modèle de *site* disponible au préalable sur chaque machine.

Déploiement

Les activités du déploiement d'une application se résument aux processus suivants : la sélection des composants logiciels, l'installation et la désinstallation. **ORYA** s'appuie sur certaines technologies dites des procédés et de fédération développées par

les mêmes équipes. Ces différentes technologies permettent de donner une description de différents processus de déploiement. Un éditeur est proposé qui va permettre de décrire graphiquement ces processus. L'exécution du processus de déploiement s'effectue de façon centralisée par un gestionnaire de déploiement. Ce dernier interagit en mode client/serveur avec des nœuds serveurs d'applications pour rapatrier les applications appropriées vers les nœuds cibles selon leurs caractéristiques matérielles et logicielles.

Gestion de l'hétérogénéité

Pour remédier au problème d'hétérogénéité, **ORYA** se base sur la notion de famille de logiciels. Une famille de logiciels correspond à un ensemble de versions d'une même entité logicielle qui peut être déployé sur les mêmes types d'environnement. Le processus de déploiement dans **ORYA** passe par une phase dite de sélection. Cette phase consiste à créer les familles de logiciels (par exemple une compilation de DIET pour le même type de processeur, système d'exploitation, etc.). Par composition de ces familles, on peut construire une application (ex composition de LA, MA, SeD forme une application DIET). Après la construction de l'application, la phase d'installation peut débuter. Pour trouver une version d'application compatible avec une machine, la description de cette dernière est utilisée. La description est utilisée pour vérifier qu'une application peut être installée sur la machine. En effet, toute application nécessite une configuration matérielle minimale pour pouvoir s'installer ou s'exécuter sur une machine. Cette configuration est comparée avec la configuration de la machine pour valider le déploiement. Par ailleurs le choix de la configuration de l'application peut être établi à partir de la configuration d'une machine. La description matérielle est donc utilisée soit pour établir la configuration souhaitée, soit pour valider la compatibilité d'une application avec une machine.

Tolérance aux pannes

ORYA dispose de capacités de synchronisation en cas de problèmes lors du déploiement. Ainsi en cas de panne, par exemple en cas de déconnexion impromptue d'une machine du réseau, **ORYA** est capable de rétablir l'état de la machine avant la panne. Dans le contexte du déploiement à grande échelle un tel comportement est important.

Synthèse

ORYA est un bon outil de déploiement d'applications au sein d'une entreprise. Il fournit un excellent support pour la gestion du déploiement. Son approche permet de structurer et d'automatiser les différentes étapes du déploiement. Il permet également de lier les concepts des différents domaines utilisés dans le déploiement. Il a par ailleurs ses limites : le déploiement est effectué à partir d'une machine centrale ce qui peut engendrer un problème de passage à l'échelle. Le processus de déploiement dans **ORYA** n'est pas figé. L'administrateur peut décrire ses propres opérations à exécuter pour déployer son application.

4.1.3 GoDIET

Description générale

GoDIET [CCD06] est un outil de déploiement de l'application DIET et des services associés sur la grille. Il a été développé par l'équipe GRAAL à l'ENS de Lyon. Rappelons que **DIET** (**D**istributed **I**nteractive **E**ngineering **T**oolbox) est une application distribuée composée de deux types d'agents : MA (Master Agent), LA (Local Agent) et des serveurs : SeD (Server Daemon). Les agents sont organisés de manière arborescente. Les MA sont au sommet de l'arbre suivis des LA et SeD. Pour déployer une application DIET, **GoDIET** prend en entrée un fichier XML contenant à la fois la description de l'infrastructure matérielle, l'emplacement des binaires et bibliothèques sur les nœuds, ainsi que la description en extension de la hiérarchie DIET à déployer. Il ne réalise pas le transfert des binaires et librairies, mais configure et lance de façon ordonnée la hiérarchie. Les caractéristiques visées par cet outil sont la portabilité, l'extensibilité. Une interface console ou graphique peut être utilisée pour déployer. Pour assurer la portabilité, **GoDIET** a été implanté en java.

Expressivité

Pour déployer une architecture DIET, **GoDIET** propose de décrire les agents dans un fichier basé sur le formalisme XML (figure 4.3).

Ce fichier de description est constitué de trois parties principales. La première partie, *ressources*, présente la description de l'infrastructure matérielle. Elle contient la description des machines hôtes utilisées pour le déploiement, et leur protocole d'accès. La deuxième partie, *diet_services*, contient les informations pour le déploiement des services de base comme le service de *nommage de corba*, mais aussi des agents de journalisation, chargés de collecter les informations sur l'application. La troisième partie, *diet_hierarchy*, permet d'affecter, pour chaque agent (MA, LA ou SeD), la machine hôte sur laquelle il est déployé, et le cas échéant l'exécutable à lancer.

Déploiement

GoDIET s'exécute sur une machine centrale et se charge d'effectuer les tâches de déploiement sur les machines distantes. Le déploiement consiste juste à générer les fichiers de configuration des entités (agents) et de les lancer selon un ordre approprié. Les fichiers binaires sont préalablement installés sur les machines. Le transfert des binaires n'est donc pas effectué par **GoDIET**.

Gestion de l'hétérogénéité

La gestion de l'hétérogénéité n'est pas prise en compte. Comme évoqué précédemment, les fichiers binaires avec les libraires sont préalablement installés sur les

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE diet_configuration SYSTEM "devel/GoDIET-2.0.0/GoDIET.dtd">
<diet_configuration>
  <goDiet debug="1" saveStdOut="no" saveStdErr="no" useUniqueDirs="no"/>
  <resources>
    <scratch dir="/homePath/user/scratch_godiet"/>
    <storage label="g5kBordeauxDisk">
      <scratch dir="/homePath/user/scratch_runtime"/>
      <scp server="frontale.bordeaux.grid5000.fr" login="pkchouhan"/>
    </storage>
    .
    .
    <storage label="g5kOrsayDisk">
      <scratch dir="/homePath/user/scratch_runtime"/>
      <scp server="frontale.orsay.grid5000.fr" login="pkchouhan"/>
    </storage>
    <cluster label="g5kBordo" disk="g5kBordeauxDisk" login="pkchouhan">
      <env path="/homePath/user/demo/bin" LD_LIBRARY_PATH="/homePath/user/demo/lib"/>
      <node label="node-1.bordeaux.grid5000.fr">
        <ssh server="node-1.bordeaux.grid5000.fr"/>
      </node>
              .
              .
      <node label="node-47.bordeaux.grid5000.fr">
           <ssh server="node-47.bordeaux.grid5000.fr"/>
      </node>
    </cluster>
         .
         .
    <cluster label="g5kOrsay" ..>
          .
          .
    </cluster>
  </resources>
  <diet_services>
    <omni_names contact="gdx0005.orsay.grid5000.fr" port="2809">
        <config server="gdx0005.orsay.grid5000.fr" remote_binary="omniNames"/>
    </omni_names>
    <log_central>
        <config server="gdx0007.orsay.grid5000.fr" remote_binary="LogCentral"/>
    </log_central>
    <log_tool>
        <config server="gdx0007.orsay.grid5000.fr" remote_binary="DIETLogTool"/>
    </log_tool>
  </diet_services>
  <diet_hierarchy>
    <master_agent label="MA1">
      <config server="node-5.toulouse.grid5000.fr" remote_binary="dietAgent"/>
      <local_agent label="LA1">
        <config server="node-75.sophia.grid5000.fr" remote_binary="dietAgent"/>
        <SeD label="SeD1">
            <config server="node-65.sophia.grid5000.fr" remote_binary="BLASserver"/>
        </SeD>
            .
            .
        <SeD label="SeD89">
            <config server="node-34.sophia.grid5000.fr" remote_binary="BLASserver"/>
        </SeD>
      </local_agent>
        .
        .
      <local_agent label="LA8">
        <config server="node-30.lyon.grid5000.fr"
                .
                .
      </local_agent>
    </master_agent>
  </diet_hierarchy>
```

FIGURE 4.3 – Extrait d'un fichier de description de **GoDIET**

nœuds avant même que **GoDIET** n'entame le déploiement. L'administrateur est donc chargé d'installer les bons binaires compatibles aux nœuds avant d'utiliser **GoDIET** pour effectuer le lancement.

Tolérance aux pannes

Pour surveiller les entités logicielles déployées, **GoDIET** propose une approche basée sur la génération des fichiers de logs (traces) en utilisant le système **LogService** (Figure 4.4). Cela consiste à récupérer les sorties standard et erreurs qui sont générées par les agents DIET (MA,LA SeD). Ces différentes sorties peuvent être utilisées pour détecter les pannes et réparer les éventuelles erreurs lors du déploiement. Rappelons que **LogService** est un système de surveillance qui relaye les messages et les informations qui surviennent dans une plate-forme distribuée. Il s'agit d'un système relativement générique qui doit être interfacé avec l'application à surveiller. Pour cela chaque élément à surveiller se voit attaché un gestionnaire spécifique à **LogService**, qui relate les évènements à un autre gestionnaire centralisé. Celui-ci récupère tous les messages d'erreurs produits par les agents.

Lorsqu'une panne survient au cours du déploiement d'une architecture DIET, **GoDIET** suspend le déploiement de tout élément inférieur à l'entité défectueuse dans la hiérarchie. Cela a pour objectif d'éviter d'autres erreurs probables. Après la suspension de l'entité défectueuse, son état est marqué comme étant en panne et **GoDIET** attend une intervention de l'administrateur afin de savoir si le déploiement de toute la hiérarchie est en danger ou non. Par ailleurs, lorsqu'une panne survient en cours d'exécution d'une application, **LogService** est chargé simplement de détecter la panne sans possibilité d'entreprendre une réparation. Cette tâche est réservée à l'administrateur qui doit réparer manuellement l'entité défaillante (par exemple la redéployer).

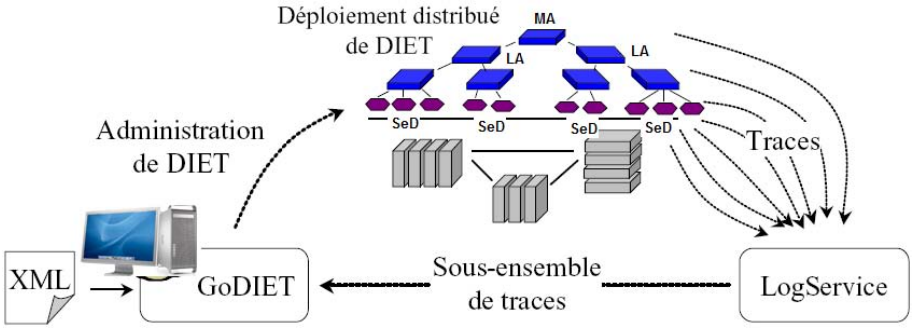

FIGURE 4.4 – Architecture de **GoDIET** avec le **LogService**

Synthèse

GoDIET est conçu exclusivement pour déployer une architecture DIET. Il configure et lance la hiérarchie de DIET de façon ordonnée sans effectuer le transfert de paquetages. Il propose un formalisme basé sur le langage XML pour la description des agents et l'infrastructure matérielle. La hiérarchie de DIET, les paramètres de configuration de chaque agent de la hiérarchie, les machines utilisées pour le déploiement sont tous décrits dans un seul et un unique fichier XML. Cela engendre un problème de modification permanente de la description lorsqu'on change d'environnement de déploiement. En effet, le fichier de description contient à la fois la description matérielle et logicielle. A cet effet, quand on change d'environnement de déploiement il est nécessaire de modifier le fichier de description. Or en séparant les

deux descriptions (matérielles et logicielles), le changement d'environnement nécessite juste la modification de la description de l'infrastructure matérielle. Le fichier de description pose également un problème de passage à l'échelle. Par exemple le déploiement de 1000 SeD nécessite un fichier de description de plus de 3000 lignes.

Il faut noter par ailleurs qu'un utilitaire graphique **XMLGoDIETGenerator** [gra0xb] simplifie la génération de la description de l'application en ne demandant qu'un minimum d'informations pour déployer une architecture DIET.

4.1.4 SmartFrog

Description générale

SmartFrog (**Smart Fr***amework for* **o***bject* **g***roups*) [GGL+03] [Sab06] est un framework conçu pour le déploiement et la gestion d'applications réparties s'exécutant sur une grappe de machines. Il a été développé par HP Labs à Bristol. L'objectif de cet outil est la gestion du cycle de vie d'une application. Pour rendre le déploiement plus souple, l'application est décrite en utilisant un ADL (Architecture Desciption Language) spécifique. Pour effectuer le déploiement d'un composant, celui-ci doit respecter un modèle bien précis spécifié par **SmartFrog**. Pour les applications patrimoniales, elles doivent être encapsulées dans des composants suivant le modèle **SmartFrog**. Les machines utilisées par **SmartFrog** pour le déploiement doivent héberger au préalable un démon spécifique.

Expressivité

SmartFrog ne dispose pas de formalisme de description des caractéristiques et de la topologie de l'environnement matériel. Cependant il dispose d'un langage déclaratif pour la description de l'application. Il permet de décrire l'architecture du système à déployer. Un système est constitué de plusieurs composants. Chaque composant de l'application est décrit avec les paramètres de configuration et le nœud de déploiement. En associant à chaque composant lors de la description, crée une dépendance entre la description de l'application et l'environnement de déploiement. Le changement d'environnement de déploiement nécessite une modification de la description de l'application.

SmartFrog dispose d'un formalisme pour décrire les étapes du déploiement et des primitives pour effectuer le déploiement de façon séquentielle ou parallèle.

La figure 4.5 suivante montre la description d'un système composé de deux serveurs web et une base de données.

Les composants qui sont à gauche de la figure (4.5) sont appelés des *templates*. Ceux-ci permettent de définir une description générique des composants pouvant

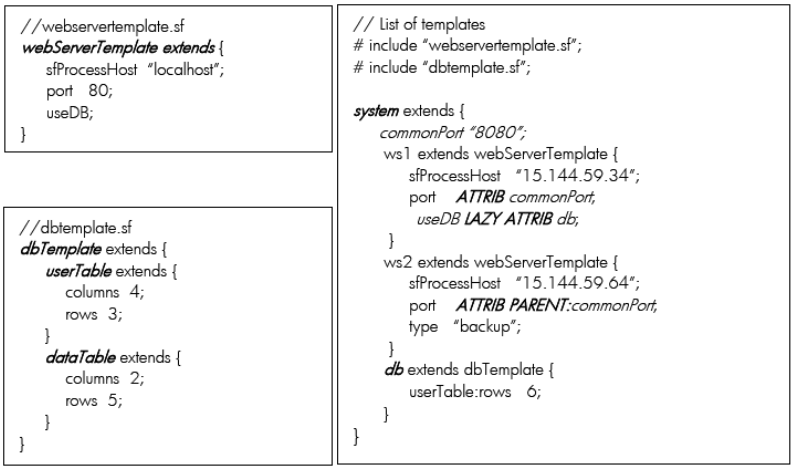

FIGURE 4.5 – Description d'une application **smarFrog**

être utilisée ensuite dans la description de la structure du système final. La description d'une entité logicielle peut hériter des caractéristiques d'un *template*. Cela peut simplifier la tâche de description aux administrateurs.

Déploiement

Pour déployer une application, **smarFrog** utilise un programme principal qui s'exécute sur une machine centrale et de plusieurs démons distribués sur les nœuds de l'infrastructure matérielle. Tout le processus de déploiement est orchestré de façon centralisée. Les démons sont chargés de l'exécution effective des tâches sur les machines et de fournir un environnement de déploiement et d'exécution des différents composants **SmartFrog**. Le gestionnaire du déploiement dans **SmartFrog** est dénommé *SFInstaller*. Ce dernier est chargé de déploiement de démons ainsi que les copies des ressources en utilisant le protocole *scp* ou *ftp*. Pour faire interagir les démons entre eux, un autre démon bien spécifique appelé *coordinateur* est désigné. Le *coordinateur* est chargé de coordonner la communication entre les démons.

SmartFrog fournit des interfaces pour interagir avec un composant logiciel lors du déploiement. Trois opérations sont exposées : *Deploy*, *Start* et *Terminate*. La seule interaction avec les composants logiciels se limite à ces trois opérations. La figure 4.6 contient deux schémas : le premier schéma montre un composant **SmartFrog** ; le deuxième explique le cycle de vie d'un composant **SmartFrog** et l'interface de programmation que ces composants doivent fournir.

Gestion de l'hétérogénéité

La gestion d'hétérogénéité n'est pas effectuée automatiquement. En effet l'administrateur est chargé d'associer manuellement lors de la description de l'application, les nœuds aux fichiers binaires de l'application qui vont être exécutés sur ces der-

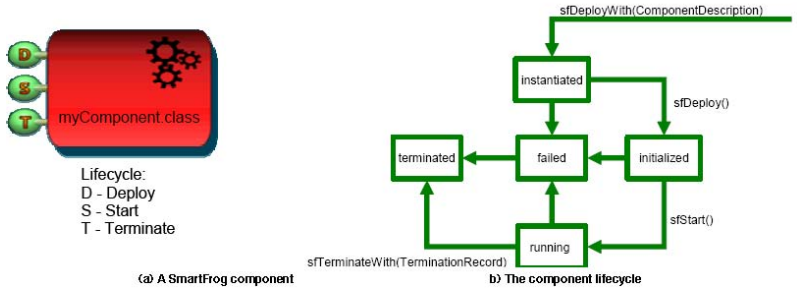

FIGURE 4.6 – Modèle de composant **SmartFrog** et son cycle de vie

niers. Le gestionnaire du déploiement est chargé de récupérer les logiciels dans un entrepôt puis entamer les procédures d'installation.

Tolérance aux pannes

Il existe un mécanisme permettant la tolérance aux fautes. Les démons sont capables de détecter les pannes. Lorsqu'un nœud tombe en panne, **SmartFrog** l'exclut alors du groupe et permet juste le redéploiement des applications qui s'exécutaient sur le nœud fautif. Cela est rendu possible grâce à la connaissance partagée de la description de l'application et de ce qui existe effectivement sur les différentes machines.

Synthèse

SmartFrog est un outil de déploiement distribué. Il dispose d'un ensemble de démons répartis pour exécuter les tâches de déploiement sur les machines physiques et un programme (*SFInstaller*) pour exécuter les opérations distantes. **SmartFrog** dispose d'un formalisme qui permet de décrire d'une part les paramètres de configuration d'une application et l'interconnexion entre ses entités logicielles. La détection de pannes matérielles est confiée aux démons. Ces derniers s'exécutent sur les nœuds durant tout le cycle de vie de l'application. Ainsi en cas de panne d'un nœud, il est exclu du groupe de démons (le démon qui tournait sur le nœud fautif). L'administrateur peut décider ou pas de redéployer les entités contenues sur le nœud fautif. La gestion d'hétérogénéité est effectuée manuellement par l'administrateur qui se charge d'associer lors de la description à chaque nœud la version de l'application qui lui est compatible. **SmartFrog** comporte quelques inconvénients. Les fonctions de reconfiguration sont limitées aux opérations : *Deploy*, *Start* et *Terminate*. Le coût de la communication entre les démons peut être exponentiel. Cela peut poser un problème de passage à l'échelle. **SmartFrog** ne propose pas de formalisme pour décrire la topologie et les caractéristiques de l'infrastructure matérielle. Malgré la répartition des démons, toutes les tâches distantes sont orchestrées et exécutées par le programme **SFInstaller**.

4.1.5 Software Dock

Description générale

Software Dock (SD) [HHW99] [Hal99] est un outil de déploiement permettant d'installer et d'administrer les applications de type *client/serveur*. Il est le résultat d'un travail de recherche effectué à l'université du Colorado. Dans **Software Dock**, il y a 2 concepts importants : les *Docks*, qui modélisent les *clients/serveurs* et les fichiers *DSD (Deployable Software Description)*, qui décrivent les applications dans un formalisme basé sur XML. L'approche de SD est basée sur un système multi-agents. Les agents sont chargés de réaliser les différentes tâches de déploiement. SD effectue le déploiement des logiciels d'une entreprise en introduisant trois niveaux d'exécution : *site* qui regroupe un ensemble de machines utilisées par les mêmes types de clients ou d'utilisateurs (machines pour le service de comptabilité, d'achat, etc.) ; *producteur* qui produit le logiciel et orchestre son déploiement ; *entreprise* formée par un ensemble de *sites*. Nous retrouvons trois types de *Docks* s'exécutant sur les trois niveaux d'exécution précédemment cités : **FieldDock** , **ReleaseDock**, **InterDock** .

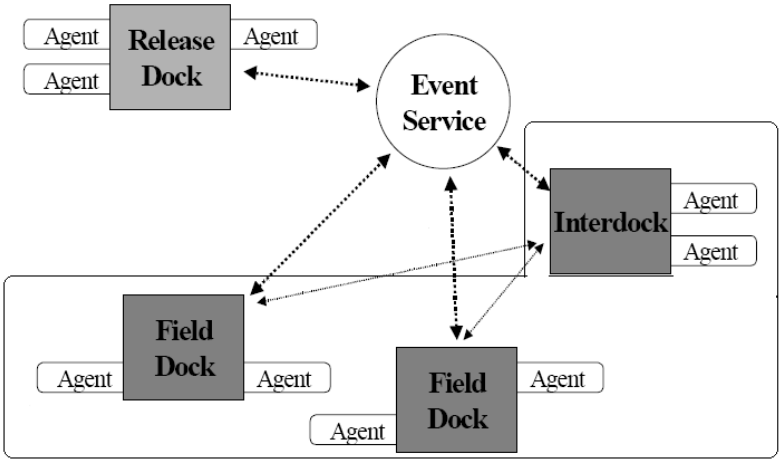

FIGURE 4.7 – Architecture de **Software Dock**

FieldDock : Il s'exécute au niveau des *sites* sur lesquels les applications seront déployées. Ce Dock permet de retrouver des informations sur le *site* lui-même (caractéristiques matérielles) et sur les applications déjà déployées. Chaque **FieldDock** (il y en a un par *site*) est chargé de gérer (via des agents) les modifications imposées par l'environnement, comme l'installation ou la suppression d'une application.

ReleaseDock : Il s'exécute au niveau du *producteur* logiciel. Il peut être considéré à la fois comme un gestionnaire de dépôts de logiciels et du processus de déploiement. Il permet à l'environnement d'avoir connaissance des différentes applications disponibles pour le déploiement. C'est lui qui est chargé de réaliser les différentes activités du déploiement (installation, configuration, etc.) via ses agents.

InterDock : Il réside dans une entreprise regroupant un ensemble de *sites* clients. Il sert d'intermédiaire entre **FieldDock** et **ReleaseDock**. Il comporte un certain nombre d'informations permettant d'avoir une vue globale de l'entreprise notamment les fichiers de description des applications et de l'environnement.

Ces trois *Docks* sont mis en relation grâce à un service d'événement **Event Service** chargé de jouer le rôle de médiateur.

Expressivité

Pour décrire la structure et la configuration de l'application à déployer, **Software Dock** utilise le formalisme DSD basé sur le langage XML. Ce formalisme permet de décrire une application et ses relations avec d'autres applications. Une application est décrite en terme de propriétés représentant ses paramètres de configuration, de contraintes qui décrivent par exemple le système d'exploitation et le type d'architecture matérielle compatible à l'application, et les fichiers nécessaires au fonctionnement de l'application.

Software Dock ne propose pas de décrire la structure et l'architecture d'une application. Cependant grâce aux contraintes, on peut spécifier les dépendances d'une application. Cela a une limite car pour déployer une application avec plusieurs entités, il faut déployer chaque entité indépendamment puis spécifier ses dépendances sous forme des contraintes dans son fichier de description. Cette approche cache également la structure globale de l'application.

Software Dock offre un formalisme pour décrire les *sites*. Il ne propose cependant pas une description de la structure globale de l'infrastructure matérielle.

Déploiement

C'est le **ReleaseDock** qui est chargé d'orchestrer le processus de déploiement de façon centralisée. Il représente le cœur du **Software Dock**. Il propose des agents pour exécuter les opérations de déploiement. En effet **ReleaseDock** dispose de plusieurs agents qui sont utilisés lors du déploiement pour l'exécution effective des tâches sur les nœuds physiques. Par exemple pour installer un logiciel sur le *site* d'un utilisateur, un agent est envoyé sur ce dernier. Cet agent a pour rôle de collecter les informations sur les configurations matérielles et logicielles. Il installe d'abord le gestionnaire **FieldDock**. Ce dernier permet d'accéder aux informations des ressources d'un site (caractéristiques matérielles) et sur les applications déjà déployées. Ensuite, l'agent utilise **FieldDock** pour collecter les informations. Selon les informations collectées, le logiciel compatible est sélectionné puis installé sur le *site*.

Gestion de l'hétérogénéité

Pour gérer l'hétérogénéité, chaque application décrit les configurations matérielles nécessaires à son fonctionnement sous forme de contraintes. En effet une application

est décrite en termes de propriétés et de contraintes. Les propriétés permettent de décrire l'application elle-même (numéro de version, nom de l'application, etc.). Les contraintes correspondent aux exigences (matérielles et logicielles) de l'application afin de pouvoir fonctionner correctement. Ainsi pour déployer une application sur un *site*, **Software Dock** la configure selon les caractéristiques matérielles et logicielles fournies par le descripteur du *site*.

Tolérance aux pannes

Une fois l'application installée, **Software Dock** offre des outils de gestion qui permettent de gérer les différentes applications présentes sur un *site* et de gérer les différentes machines de l'entreprise. Cette gestion implique la mise à jour, l'installation, la reconfiguration et la désinstallation. Pour cela **Software Dock** enregistre la structure de chaque application présente sur le *site*. Ainsi en cas de problème (panne logicielle) il est possible de s'en apercevoir et de rétablir la cohérence de l'application.

Synthèse

Software Dock fournit une solution qui s'adresse à l'ensemble du cycle de vie du déploiement. Son architecture est distribuée et composée de trois gestionnaires principaux (**ReleaseDock**, **FieldDock** et **InterDock**) et un gestionnaire d'événements (**Event Service**) pour faire communiquer les trois autres gestionnaires. Le gestionnaire **ReleaseDock** orchestre le processus de déploiement et **FieldDock** permet de collecter les informations de l'infrastructure matérielle. **ReleaseDock** utilise des agents pour exécuter les tâches du déploiement sur les nœuds physiques. Pour décrire la configuration de l'application à déployer, **Software Dock** utilise le langage *DSD* basé sur XML. Le choix d'une application pour un *site* est effectué en fonction de la description de ce dernier. Cela permet au **Software Dock** de remédier au problème d'hétérogénéité. Cependant **Software Dock** a des points faibles. En effet, il ne permet pas une description de l'architecture logicielle d'une application et propose un processus de déploiement figé et non modifiable, ce qui empêche de l'adapter aux différents contextes de déploiement. Malgré la répartition des gestionnaires sur plusieurs sites, le cœur du **Software Dock** est basé sur **ReleaseDock** qui orchestre le processus de déploiement de façon centralisée.

4.1.6 Taktuk

Description générale

Taktuk [MR03] [Mar04] [CHR09] est un outil de déploiement d'applications parallèles pour les clusters de grande taille. Il a été développé en France au laboratoire **LIG** (**L**aboratoire d'**I**nformatique de **G**renoble). Il peut être utilisé pour effectuer des opérations d'administration en fournissant uniquement un service d'exécution distante parallèle, hiérarchique et performante. L'administration est limitée à une

simple exécution de commandes ou de scripts. L'exécution distante dans **Taktuk** est indépendante du protocole (*rsh, ssh, ssf, rexec*, etc...) utilisé. **Taktuk** est capable de traiter des plateformes aux configurations hétérogènes, utilisant des protocoles d'exécution distante différents suivant les nœuds cibles.

Expressivité

Taktuk propose un service d'exécution de commandes/scripts sur un ensemble de machines de façon hiérarchique. A cet effet, il ne propose ni un formalisme de description d'applications ni un formalisme pour décrire l'infrastructure matérielle.

Déploiement

Le déploiement dans **Taktuk** est effectué de manière hiérarchique. Son approche est basée sur la participation d'un ensemble de nœuds au déploiement. Pour assurer le passage à l'échelle, l'administrateur a la possibilité de désigner plusieurs nœuds qui peuvent participer au déploiement permettant ainsi de distribuer les tâches. Pour chaque nœud qui participe au déploiement (appelons *nœud déployeur*), on donne une liste de nœuds sur lesquels le *nœud déployeur* est chargé d'effectuer le déploiement. Tout nouveau *nœud déployeur* atteint par le déploiement participe à son tour au déploiement sur sa liste de nœud. Cela permet de distribuer le travail total (nombre total des appels distants nécessaires) sur plusieurs nœuds. Il résulte de ce déploiement *récursif* un arbre de déploiement couvrant l'ensemble des nœuds cibles. La topologie de cet arbre est très importante pour les performances du déploiement, car elle correspond à l'ordonnancement des différents appels distants. La figure 4.8 montre le déploiement d'un programme sur plusieurs nœuds. Le nœud de nom *host1* est chargé d'exécuter le programme sur les nœuds (*host1-1, host1-2,... host1-m*). Le nœud de nom *host2* est chargé d'exécuter sur les nœuds (*host2-1, host2-2,... host2-n*). Ainsi de suite jusqu'au nœud *host3* qui ne participe pas au déploiement.

```
taktuk -m host1 -[ -m host1-1 ... -m host1-m -]
       -m host2 -[ -m host2-1 ... -m host2-n -]
       -m host3 broadcast exec [hostname]
```

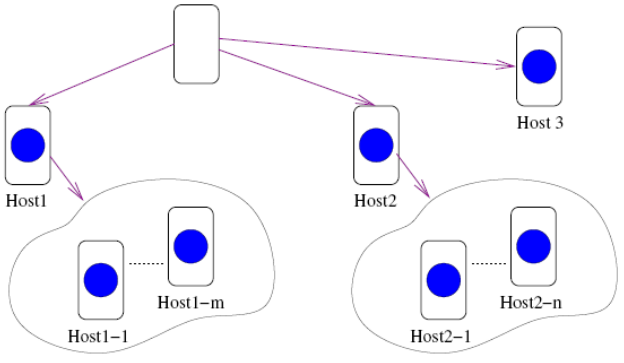

FIGURE 4.8 – Déploiement hiérarchique avec **Taktuk**

Gestion de l'hétérogénéité

La gestion d'hétérogénéité est effectuée manuellement par l'utilisateur. Pour qu'une commande soit exécutée correctement sur une machine, il faut que cette dernière soit conforme par exemple au type du système d'exploitation de la machine. Un exemple concret est la mise en place d'une variable d'environnement qui varie selon le type du système d'exploitation de la machine (*set* pour WINDOWS, *export* ou *setenv* sur un système de type UNIX/LINUX).

Tolérance aux pannes

Le mécanisme de tolérance aux pannes se limite à la détection de nœuds non atteignables lors du processus de déploiement. L'approche de **Taktuk** est basée sur un temps de tentative de connexion qui ne doit pas être dépassé. Cela consiste à limiter le temps d'attente de réponse si le nœud n'est pas atteignable. Tout nœud non atteint à l'issue du temps d'attente est considéré comme défaillant. Le choix du temps est déterminant pour cette approche. Un temps petit permet une détection de panne très rapide (au risque de considérer à tort des nœuds comme défaillants). A l'inverse un temps élevé donne un comportement plus lent mais plus soucieux de l'état réel des nœuds (garantissant une forte confiance lorsqu'une défaillance est suspectée).

Synthèse

TakTuk est un outil de déploiement à grande échelle conçu principalement pour l'exécution des commandes de façon hiérarchique sur plusieurs centaines voire des milliers de machines. Il utilise une combinaison de parallélisation (pour une efficacité maximale) et de répartition des tâches pour passer à l'échelle. Cela permet de déployer l'exécution d'une commande en un temps raisonnable. Cependant les services de déploiement sont très limités. **Taktuk** offre uniquement un service d'exécution de commandes et de scripts.

4.2 Systèmes autonomes

Dans la section précédente, nous avons présenté les outils de déploiement. Parmi ces outils notamment **SmartForg**, **Software Dock**. l'intervention d'un administrateur est indispensable pour le bon fonctionnement de l'administration d'une application. Une grande partie des tâches d'administration n'est pas formalisée et dépend plutôt du savoir-faire de l'administrateur et de son expérience. Avec la complexité croissante des systèmes et des applications, le coût de l'administration devient de plus en plus considérable et les difficultés de l'administration commencent à dépasser les capacités de traitement par des humains. Par conséquent, une nouvelle tendance est apparue qui consiste à rendre les fonctions d'administration autonomes permettant une auto-administration. Cela inclut l'*auto-configuration* (configuration automatique

suivant des règles prédéfinies), l'*auto-optimisation* (contrôle et adaptation du système pour assurer un certain niveau de performance), l'*auto-réparation* (détection de pannes et correction automatique) et l'auto-protection (prise de mesures nécessaires pour se protéger des attaques malveillantes et savoir se défendre contre ces attaques). C'est l'objectif de l'*autonomic computing* [GC03]. Dans cette section nous présentons deux outils d'administration autonome. **DeployWare** [FDDM08] est une approche qui est spécialisée dans l'administration d'applications patrimoniales et qui repose sur le modèle à composants Fractal [BCL+06a]. **JADE** [BBH+05] est un canevas d'administration autonome d'applications réparties qui est basé, comme **DeployWare**, sur le modèle à composants Fractal.

4.2.1 DeployWare

Description générale

DeployWare [FDDM08] [FM06] [DFDM08] aussi connu sous le nom de **FDF** (**F***ractal* **D***eployment* **F***ramework*) est un canevas générique développé au LIFL à Lille en France en 2005. L'objectif principal de cet outil est l'administration des systèmes distribués autonomiques. Ce travail de recherche récent, abstrait chaque notion du processus d'administration sous la forme d'un composant et fournit une manière d'exécuter le déploiement. Des composants autonomes peuvent être définis pour effectuer une administration autonome. **DeployWare** repose sur le modèle à composants Fractal et propose l'administration des logiciels sur divers types d'infrastructures matérielles telle que la grille. Il propose également un langage spécifique au domaine du déploiement (un DSL pour le déploiement) reposant sur un méta-modèle capturant les concepts abstraits d'administration et une machine virtuelle pour ce langage, une bibliothèque de composants proposant des services pour le déploiement, la reconfiguration et une console graphique d'administration appelée **Deployware eXplorer**.

DeployWare définit trois rôles dans l'administration d'un logiciel : *expert logiciel* est chargé de définir le processus de déploiement, c'est le spécialiste de la technologie du logiciel à déployer ; *administrateur système* donne les configurations réseaux (description de l'infrastructure matérielle de déploiement) ; *utilisateur final* utilise la console graphique d'administration (**DeployWare eXplorer**) pour administrer son application.

Expressivité

Comme évoqué précédemment, **Deployware** propose un langage spécifique pour le déploiement (un DSL) et une machine virtuelle pour ce langage. Le langage **DeployWare** est défini par un méta-modèle, et propose une notation graphique sous la forme d'un profil UML. Ce langage permet de décrire les entités logicielles d'une application et sera interprété par la machine virtuelle **FDF**. **FDF** est implanté sous

la forme de composants Fractal réifiant les logiciels à déployer et l'infrastructure matérielle. Pour déployer un logiciel, il doit être wrappé dans un composant Fractal. Un formalisme est utilisé pour décrire ces wrappers. Les auteurs proposent un formalisme basé sur une extension de Fractal ADL. La syntaxe XML de Fractal ADL est abandonnée au profit d'une syntaxe déclarative moins verbeuse. Ce langage a pour rôle de cacher les notions Fractal permettant ainsi aux administrateurs d'administrer leurs logiciels sans avoir connaissance du modèle à composants Fractal. La figue 4.9 montre une transformation d'un composant Fractal en langage **DeployWare**.

Par ailleurs, il est possible de décrire le opérations à effectuer pour déployer une application. Cela permet à un administrateur de contrôler le processus de déploiement de son application.

```
<definition name="MyDefinition" extends
    ="MySuperDefinition">
  <component name="a" definition="
      MyAComponent(a_value)"/>
  <component name="b">
    <component name="c" definition="
        MyCComponent"/>
    <component name="a" definition="./a
        "/>
  </component>
</definition>
```

```
MyDefinition = MySuperDefinition
{
  a = MyAComponent(a_value) ;
  b {
    c = MyCComponent ;
    a = ./ ;
  }
}
```

FIGURE 4.9 – Transformation d'un fichier Fractal ADL en langage **DeployWare**

Pour introduire une description en intension, les auteurs ont introduit le constructeur *foreach* [RM07b] de l'extension de Fractal ADL. L'objectif est de pouvoir déployer une même entité logicielle sur plusieurs nœuds.

Déploiement

Pour effectuer le déploiement, les auteurs proposent une machine virtuelle. Cette machine représente le gestionnaire de déploiement d'un logiciel. Elle prend en entrée un descripteur de déploiement et utilise les composants wrappers des entités logicielles de l'application pour exécuter le processus de déploiement sur les machines physiques.

DeployWare utilise un nombre important de composants Fractal pendant l'exécution du processus de déploiement. En effet les machines de l'infrastructure matérielle, les caractéristiques machines (exemple le login, le nom de la machine, protocole, etc.), sont encapsulées dans des composants Fractal. L'instanciation de ces multiples composants représente une charge pour la machine qui effectue le déploiement. Cela pose un problème de passage à l'échelle vu que le nombre de machines est important dans le contexte de grande échelle et chaque machine peut avoir un nombre quelconque de caractéristiques.

Pour remédier à ce problème de passage à l'échelle, les auteurs proposent de mettre en place un mécanisme de distribution de la machine virtuelle **FDF**. Cette approche permet de décentraliser le déploiement. Des gestionnaires de déploiement

intermédiaires sont introduits. L'approche est basée sur un ensemble de serveurs **FDF** déployés par **FDF** (auto-déploiement). Ces serveurs sont chargés d'exécuter le processus d'administration. Un gestionnaire dénommé *DeployWare Master* (*DW Master*) représente le serveur principal. D'autres gestionnaires appelés *DeployWare Server* (*DW Server*) représentent des serveurs auxiliaires et sont chargés pour exécuter les tâches de déploiement déléguées par *DW Master*. Pour commencer le déploiement, *DeployWare Master* déploie tout d'abord les serveurs auxiliaires *DW Server* puis il délègue à ces serveurs les tâches d'administration.

Gestion de l'hétérogénéité

Pour administrer une application avec **DeployWare**, l'administrateur fournit dans le fichier de description de chaque entité logicielle, son paquetage et le nœud de déploiement. L'administrateur doit donc avoir une connaissance du type de paquetage compatible avec un nœud. La gestion de l'hétérogénéité n'est en effet, pas automatiquement prise en compte. L'administrateur doit s'assurer de la compatibilité entre le paquetage d'une entité logicielle et son nœud de déploiement pour le bon fonctionnement du processus d'administration.

Tolérance aux pannes

Le problème de panne est résolu grâce à l'administration autonome. En effet les auteurs proposent des solutions logicielles spécifiques capables de prendre en charge l'exécution de boucles de contrôle. Ces boucles de contrôle permettent d'adapter la machine virtuelle aux éventuelles pannes logicielles et matérielles. Ils sont exprimées à l'aide du paradigme *Evénement-Condition-Action* ([CC95]) peuvent être injectées dans le logiciel qui gère un ensemble de composants prédéfinis dans **DeployWare** (Personnalité, composants d'accès aux machines hôtes etc.). Il est capable d'écouter des événements, de tester si ces événements déclenchent l'une des boucles de contrôle, et le cas échéant, lance l'exécution de la reconfiguration adéquate.

Synthèse

DeployWare est un outil permettant d'effectuer l'administration à grande échelle d'applications réparties. Une machine virtuelle et une console graphique sont proposées pour exprimer graphiquement le déploiement des logiciels. Cela facilite beaucoup les tâches d'administration pour les utilisateurs non expérimentés. **DeploWare** propose d'encapsuler tout en composant Fractal du simple paramètre d'un logiciel au nœud sur lequel le déploiement de l'application est effectué. Par exemple un nœud est représenté par un composite Fractal constitué de plusieurs composants primitifs (*shell, protocol, hostname, user*, etc.). Chaque composant primitif est responsable d'une tâche permettant d'accéder au nœud et d'exécuter les commandes sur ce dernier. Pour connaître le type du *shell* d'un nœud, il faut s'adresser au composant primitif *shell* du composite nœud. L'exécution d'une commande sur le nœud passe par le composant *protocol*. Le nom du nœud est également encapsulé dans un composant *hostname*, et le nom de connexion (i.e. le login) est contenu dans le composant

user. Toutes les caractéristiques d'un nœud sont encapsulées dans des composants primitifs contenus dans le composite *host* (nœud) encapsulant la machine physique. Cet ensemble de composants représente une charge pour le nœud qui déploie et peut diminuer la performance du système lorsqu'on passe à l'échelle (plusieurs nœuds et plusieurs paramètres de configuration de nœuds donc des multiples composants à instancier pendant l'administration). Une solution a été proposée pour résoudre ce problème. Cette solution est basée sur la répartition de la machine virtuelle sur plusieurs machines.

La solution proposée pour passer à l'échelle présente quelques inconvénients. En effet tout passe par le **DW Master** du coup il représente un goulot d'étranglement.

Le langage **DeployWare** ne sépare pas la description des machines et des applications. La gestion d'hétérogénéité logicielle est effectuée manuellement. L'administrateur doit associer la bonne version du paquetage au nœud d'une entité logicielle.

4.2.2 JADE

Description générale

JADE [BBH+05] [SBDP08] [TBDP+06] [TBD+06] est une plate forme développée à l'INRIA à Grenoble en France en 2004 pour l'administration autonome d'infrastructures logicielles patrimoniales. Le terme *patrimoniale* désigne ici une application vue comme une boîte noire, uniquement accessible via son interface applicative. **JADE** est essentiellement composé de deux parties : un canevas pour l'encapsulation des ressources administrées, qui leur donne une interface d'administration uniforme, et un canevas de construction de gestionnaires autonomes, qui administrent à l'exécution un ensemble de ressources suivant une politique particulière. On distingue trois grandes composantes qui orchestrent l'administration :

La représentation du système permet essentiellement de conserver une copie de l'architecture du système. Cette copie contient l'ensemble du système : les ressources administrées ainsi que l'application à administrer. La représentation du système et le système administré sont maintenus en cohérence, toute action ou changement d'état de l'un étant répercuté sur l'autre, et réciproquement.

Le système administré est composé d'un ensemble d'entités logicielles qui constituent l'application patrimoniale. Pour avoir une vision homogène de l'environnement, les entités logicielles sont encapsulées dans des composants Fractal.

Un gestionnaire fournit l'automatisation d'un aspect de l'administration. Il existe un gestionnaire prenant en charge le déploiement, un autre qui prend en charge les aspects d'allocation des machines. Il existe aussi des gestionnaires dits autonomes. Un gestionnaire autonome fournit un comportement autonome pour la

gestion de la ressource dont il a la charge. Le principe général de ces gestionnaires est de rétroagir sur le système à partir d'informations prélevées. Ce principe est communément appelé : boucle de contrôle. Un exemple de gestionnaire autonome est le gestionnaire de fautes. Les gestionnaires fournis par **JADE** sont optionnels. Leur usage dépend du besoin particulier des applications et de leurs contextes d'utilisation. L'approche à composants permet d'obtenir une extrême souplesse dans les possibilités d'adaptation et d'extension de **JADE**. Chaque gestionnaire peut être ajouté ou enlevé de façon indépendante ou bien reconfiguré afin d'implanter telle ou telle autre politique de gestion de ressource.

La figure 4.10 montre une vision globale de **JADE**. Les trois grandes composantes sont identifiables dans l'architecture de **JADE**. Nous pouvons distinguer la **représentation du système**, le **système administré** et l'ensemble des **gestionnaires**.

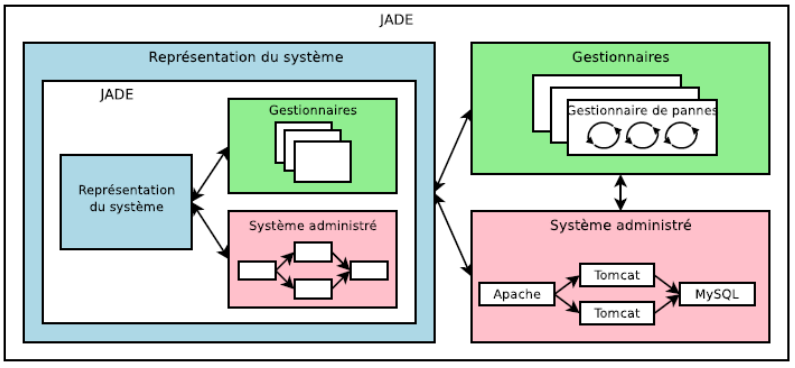

FIGURE 4.10 – Architecture de **JADE**

Expressivité

La description de l'infrastructure matérielle et logicielle est effectuée en utilisant le langage Fractal ADL. En effet le modèle à composants Fractal fournit un langage de description d'architecture appelé Fractal ADL. C'est un langage déclaratif qui permet la description de la structure de l'application construite à partir des composants Fractal. La base du langage est minimale : Fractal ADL fournit uniquement des constructions de base pour énumérer des composants, des interfaces, des liaisons et des attributs et laisse les concepteurs étendre le langage pour intégrer d'autres informations spécifiques à leur cadre d'utilisation. La syntaxe du langage Fractal ADL est basée sur XML. La figure 4.11 illustre un exemple de description d'architecture pour une application simple contenant un composant *client* qui appelle des opérations sur un composant *serveur*.

Pour faciliter la description d'architecture, Fractal ADL donne la possibilité de répartir dans plusieurs fichiers la description d'une architecture. Les relations entre ces fichiers peuvent prendre des formes différentes. Par exemple une définition peut en étendre une autre en y rajoutant certains éléments ou en surchargeant certaines propriétés.

```
<definition name="HelloWorld">
   <interface name="r" role="server"
      signature="java.lang.Runnable"/>

   <component name="client">
      <interface name="r" role="server"
         signature="java.lang.Runnable"/>
      <interface name="s" role="client"
         signature="Service"/>
      <content class="ClientImpl"/>
   </component>

   <component name="server">
      <interface name="s" role="server"
         signature="Service"/>
      <content class="ServerImpl"/>
   </component>

   <binding client="this.r" server="client.r"/>

   <binding client="client.s" server="server.s"/>

</definition>
```

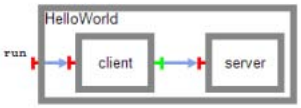

FIGURE 4.11 – Description de l'application *client/serveur* avec Fractal ADL

Déploiement

Le gestionnaire d'administration est composé de plusieurs gestionnaires autonomes (Autonomic Manager ou AM). L'administration dans **JADE** repose sur la répartition des démons ayant la charge d'exécuter les tâches élémentaires d'administration sur le nœud physique. Ces démons sont dénommés éléments administrables (Managed Element ou ME). Ces différents éléments d'administration (ME et AM) sont des composants primitifs Fractal. Pour déployer une application avec **JADE**, l'administrateur décrit dans un ou plusieurs fichiers ADL, les paramètres de configuration de chaque instance d'entité logicielle et l'architecture logicielle de l'application. Un gestionnaire de déploiement centralisé transforme cette description en une architecture à composants (ME) distribuée sur les machines cibles en utilisant le mécanisme de déploiement intégré à Fractal. Ces composants vont agir localement sur les entités logicielles patrimoniales. Un dépôt des paquetages contenant les paquetages disponibles pour installation est utilisé. Un programme déclenche le processus de déploiement en appelant les interfaces appropriées sur les composants au niveau des machines cibles.

Gestion de l'hétérogénéité

Pour gérer l'hétérogénéité des machines dans la version initiale de **JADE**, les auteurs proposent de disposer pour chaque cluster, d'une version de l'application qui lui est compatible. Ces différentes versions sont mises dans un dépôt (*repository*) et donc accessibles à partir de ce dernier. Le dépôt indique pour chaque entité logicielle et pour chaque cluster, la localisation (accessible par NFS) de la livraison binaire de ce logiciel exécutable sur ce cluster. Le contenu du dépôt est stocké dans un fichier. L'administrateur doit disposer de différentes versions et tester les logiciels sur chaque cluster, en générant différentes images binaires si nécessaire, et mettre à jour le fichier de dépôt en conséquence pour chaque cluster.

Tolérance aux pannes

La plate-forme **JADE** intègre divers gestionnaires ciblant des aspects d'administration autonome tels que l'auto-adaptation, face à des défaillances (auto-réparation) [SBDP08] [TBS+08] ou à des variations de charge (auto-optimisation) [TBD+06]. Pour remédier au problème de panne logicielle ou matérielle, le gestionnaire d'auto-réparation s'appuie sur des sondes permettant de détecter les pannes de composants logiciels ou matériels du système administré ; et sur des actionneurs permettant de redéployer et de remplacer les composants défectueux du système pour le rendre à nouveau opérationnel.

Synthèse

Le principe de représentation de l'architecture du système sous forme de composants Fractal constitue une approche efficace et novatrice. Elle permet d'abstraire la notion d'application, pour la représenter sous forme de composants en interaction. Le choix du modèle Fractal pour les composants est doublement judicieux. Les composants Fractal sont légers en terme d'exécution, et offrent des mécanismes de reconfiguration dynamique. L'architecture de composants offre un mécanisme de réflexivité pour l'administration des applications réparties.

L'expressivité dans **JADE** est basée sur le formalisme Fractal ADL. Ce formalisme est basé sur le langage XML et permet de décrire les paramètres de configuration d'une application ainsi que son architecture. L'architecture de **JADE** est distribuée. Cependant le processus d'administration est orchestré par un gestionnaire centralisé contenant la représentation à composants de l'application. La gestion de pannes est prise en compte grâce aux sondes/actionneurs qui permettent de détecter et réparer automatiquement une panne. Le problème d'hétérogénéité est résolu grâce au gestionnaire de dépôt. Ce dernier associe à chaque cluster, la version de l'application compatible aux machines.

Néanmoins, l'utilisation de **JADE** n'est pas une tâche facile. L'utilisateur doit assimiler les concepts d'architecture à composants, en plus des concepts spécifiques de Fractal, pour administrer son application. Le manque de description en intension engendre le problème de passage à l'échelle lors de la description de l'application. En effet le déploiement d'une architecture incluant un nombre important d'instances logicielles peut nécessiter un fichier de description XML de plusieurs lignes. Certes la notion d'héritage peut diminuer la taille de cette description mais elle ne résout pas le problème d'instances à décrire.

4.3 Etat de l'art : Synthèse

L'état de l'art présenté dans ce chapitre aborde différentes techniques pour l'administration d'applications réparties à grande échelle. L'administration d'une application peut être décomposée en plusieurs phases. Premièrement, déployer le ou les entités logicielles de l'application sur les machines cibles. Cette phase est décom-

posée en plusieurs sous tâches. Il faut tout d'abord décrire l'architecture logicielle et les paramètres de configuration de l'application. Ensuite installer les ressources nécessaires au démarrage de l'application (téléchargement du code binaire des entités logicielles, les librairies). Une fois les ressources téléchargées et installées, il faut réaliser toutes les opérations de configuration que l'application requiert pour son bon fonctionnement. Il faut ensuite démarrer l'exécution de l'ensemble des entités logicielles installées dans l'ordre approprié.

Ce chapitre nous a présenté plusieurs outils et techniques d´administration utilisés dans différents domaines. Les différentes techniques sont plus ou moins bien adaptées suivant le type de logiciel que l'on souhaite administrer. Dans un premier temps, nous avons étudié un ensemble de systèmes qui sont plutôt orientés vers le déploiement. Ces systèmes font également quelques tâches d'administration. Dans un second temps nous avons présenté les systèmes d'administration autonome qui effectuent les tâches d'dministration sans intervention humaine.

Nous retrouvons des systèmes ne disposant pas de notion ***d'architecture logicielle*** lors de la description de l'application notamment **Software Dock**, **ORYA**, **ADAGE**. Des systèmes comme **JADE** et **GoDIET** permettent la description de l'architecture logicielle et les paramètres de configuration de l'application. Avec **DeployWare**, l'administrateur peut décrire les paramètres de configuration des entités logicielles et énumérer leurs dépendances. Cependant, les auteurs ne proposent pas une description de l'architecture logicielle d'une application.

Nous remarquons que la ***description en intension*** est quasi inexistante dans les systèmes étudiés. Le déploiement avec **JADE**, **GoDIET** nécessite une description de chaque instance d'entité logicielle à déployer. **DeployWare** propose une primitive permettant de déployer plusieurs instances d'une même entité logicielle.

Nous remarquons également qu'hormis le système **ADAGE**, aucun autre système ne propose la ***description de la structure et la topologie d'une grille***. Les systèmes étudiés proposent généralement la description des clusters sans établir la relation entre ces derniers. **JADE** propose un fichier ADL fractal pour décrire l'ensemble des clusters utilisés lors de l'administration. **DeployWare** propose également un fichier ADL d'une extension Fractal qui contient à la fois la description des wrappers des entités logicielles à déployer et les clusters à utiliser.

Seuls **DeployWare**, **ORYA** et **SmartForg** permettent une description du processus de déploiement. Ces trois systèmes donnent la possibilité de décrire les opérations de déploiement d'une application. En effet le processus de déploiement est implanté en dur dans le code source de la plus part des outils de déploiement que nous avons étudiés. Cela engendre une limitation importante, car l'administrateur n'aura pas la possibilité de personnaliser l'enchainement des opérations de déploiement afin de tenir compte du contexte d'environnement. On ne peut donc pas modifier la politique de déploiement de ces outils, sans reconcevoir l'outil dans son ensemble.

Excepté **DeployWare** et **Taktuk**, nous remarquons que les outils que nous avons présentés orchestrent les différents processus d'administration (ou simplement du déploiement) de ***façon centralisée***. Cependant certains outils utilisent des démons distribués sur les machines pour déléguer l'exécution effective des tâches d'administration sur les nœuds physiques. Cela ne diminue pas la charge de connexions et de copies distantes sur les machines. Nous remarquons également que certains outils n'effectuent pas toutes les tâches du déploiement en l'occurrence le transfert des paquetages binaires. Parmi ces outils, nous pouvons citer **GoDiet**. En effet l'emplacement des binaires est précisé pour chaque nœud dans le fichier de description de **GoDIET**. Ainsi **GoDIET** n'effectue que le démarrage des agents et serveurs DIET.

Parmi les systèmes étudiés, la ***gestion de l'hétérogénéité*** est généralement à la charge de l'administrateur. Des systèmes notamment **ADAGE**, **Software Dock** utilisent la notion de contraintes pour remédier au problème d'hétérogénéité. En effet les besoins d'une application sont exprimés sous forme de contraintes. Lors de la description d'une application, des contraintes sont spécifiées notamment le type de système d'exploitation sur lequel l'application peut être exécutée, l'architecture processeur compatible, etc. Le système **ORYA** adopte l'approche basée sur les familles de logiciels. Une famille de logiciels correspond à un ensemble des versions d'une même entité logicielle qui peut être déployée sur les mêmes types d'environnements. **JADE** utilise un dépôt de logiciels. Il propose de disposer pour chaque cluster, d'une version de l'application qui lui est compatible.

Pour les systèmes autonomes **DeployWare** et **JADE**, la ***tolérance aux pannes*** est prise en compte par les composants autonomes. Le principe fondateur des systèmes basés sur l'approche *autonomic computing* est l'autogestion. Les pannes sont automatiquement détectées par les sondes et réparées par les actionneurs sans aucune intervention de l'administrateur. **GoDIET** se limite à une détection de panne d'un agent/serveur DIET. L'administrateur peut décider de redéployer l'agent/serveur fautif. **SmartFROG** permet de détecter et réparer les pannes d'un nœud. Lorsqu'un nœud tombe en panne, **SmartFROG** permet le redéploiement des applications qui s'exécutaient sur ce nœud.

L'analyse des différents travaux présentés dans ce chapitre sur l'état de l'art nous a permis d'identifier des concepts émergents dans le contexte de l'administration à grande échelle :

1. **Architecture logicielle** : la capacité de décrire l'architecture logicielle d'une application en vue de l'administrer.
2. **Description en intension** : la possibilité de décrire plusieurs instances d'entités logicielles sans donner la description de chaque instance qu'on veut administrer.
3. **Description de la topologie de l'infrastructure matérielle** : proposition d'un formalisme pour décrire la spécificité et la structure globale de l'environnement d'exécution de l'application.

4. **Personnalisation du processus d'installation** : donner la possibilité aux administrateurs de personnaliser, contrôler l'installation de leurs applications. Cela peut passer par la proposition d'un formalisme de description du processus d'installation.
5. **Décentralisation de l'administration ou du déploiement** : Ce concept correspond à la mise en œuvre d'une approche pour répartir la charge de l'administration ou du déploiement sur plusieurs nœuds. Cela consiste à exécuter les différentes tâches d'administration à partir de plusieurs machines. Ainsi nous parlons de la répartition ou de la décentralisation des tâches d'administration entre plusieurs systèmes d'administration. Il faut à cet effet distinguer la répartition des démons pour l'exécution effective des tâches sur les nœuds (comme dans le cas de **JADE, et SmartForg**) et la répartition des systèmes d'administration qui consiste à déployer plusieurs systèmes d'administration afin d'administrer l'application.
6. **Gestion de l'hétérogénéité** : tenir compte de l'hétérogénéité logicielle et matérielle lors du déploiement.
7. **Tolérance aux pannes** : la mise en œuvre d'un mécanisme de tolérance aux pannes lors du déploiement ou pendant l'exécution d'une application administrée. Cela peut être la détection de pannes matérielles ou logicielles, la détection du dysfonctionnement de l'exécution du processus de déploiement, etc.

Le tableau 4.1 récapitule les caractéristiques des travaux présentés sous les axes définis ci-dessus.

L'objectif de nos travaux est donc de fournir un système d'administration permettant de remédier à l'ensemble des insuffisances vues dans cet état de l'art. Nos propositions vont se porter sur l'administration à grande échelle, sur la personnalisation du processus d'installation et la gestion de l'hétérogénéité.

axes	ADAGE	ORYA	GoDIET	SmartForg	Software Dock	Taktuk	DeployWare	JADE
1	-	-	x	-	-	-	dépendance	x
2	-	-	-	-	-	-	-	-
3	x	-	-	-	-	-	-	-
4	-	x	-	x	-	-	x	-
5	-	-	-	démons	démons	x	x	démons
6	contraintes	dépôt	-	-	contraintes	-	x	dépôt
7	-	-	détection	détection/redéploiement	-	-	x	x

TABLE 4.1 – ***Tableau comparatif synthétisant les caractéristiques des systèmes étudiés. Les lignes correspondent aux concepts énumérés ci-dessus. (x) indique la présence d'un concept alors que (-) indique son absence.***

Troisième partie

Contributions

Introduction

Rappel de la problématique

Nous rappelons dans cette section les problématiques évoquées dans le chapitre 2 auxquels nous apportons des contributions. Nous commençons par présenter brièvement le contexte de la thèse. Ensuite chaque problème est résumé sous forme de rappel.

Contexte : On s'intéresse dans cette thèse à l'administration d'applications réparties dans un contexte de grande échelle tel que la grille. Une grille est un environnement complexe : ses ressources sont hétérogènes (ordinateurs et réseaux de communication), et distribuées géographiquement à grande échelle entre plusieurs localités qu'on appelle site. Administrer des applications dans ce type d'environnement est une tâche ardue et coûteuse en ressources humaines, matérielles et logicielles. Administrer une application revient tout d'abord à décrire dans un formalisme les paramètres de configuration et son architecture. Ensuite l'application est déployée puis gérée pendant son exécution. L'administration dans un contexte de grande échelle engendre des multiples problèmes à savoir :

Expressivité : Ce problème porte sur le formalisme de description de l'application. Dans un environnement grande échelle tel qu'une grille de machines, l'application à administrer peut être composée de nombreuses instances d'entités logicielles. Décrire la configuration de chacune de ces instances et leurs interactions s'avère une tâche difficile et peut engendrer des multiples erreurs de configuration. Il est nécessaire de fournir un formalisme adéquat permettant de faciliter cette tâche en tenant compte du facteur d'échelle.

Performance : L'administration d'une application répartie sur une infrastructure matérielle grille a un coût et représente une charge en terme de ressources matérielles et logicielles (mémoire, disque dur, nombre de connexions, socket, etc.). Cette charge est proportionnelle au nombre de nœuds et provient essentiellement des opérations de déploiement (copie des fichiers, génération des fichier de configuration, démarrage, etc.) et de maintenance (gestion de pannes, reconfiguration, etc.). Par exemple pour effectuer une opération de démarrage d'une entité logicielle sur des milliers de machines, il faut se connecter sur chaque machine pour exécuter le programme de démarrage. En plus, le démarrage de chaque programme nécessite la création d'un ou plusieurs processus sur la machine qui administre et l'utilisation des ressources réseaux pour les connexions sur les machines distantes. Une approche qui consiste à administrer toutes les entités logicielles à partir d'une seule machine qu'on appelle **administration centralisée**, peut engendrer plusieurs problèmes. La machine qui administre donc qui exécute les multiples opérations d'administration verra sa consommation des ressources augmenter due : aux coûts de la communication entre la machine qui administre et les machines sur lesquelles on administre ; à la création des processus pour l'exécution des opérations d'administration. Cela peut entraîner une dégradation de la performances du système d'administration.

Hétérogénéité : L'administration à grande échelle d'une application répartie considère de milliers de machines distribuées sur des sites très hétérogènes. Par exemple les machines des sites peuvent fonctionner sous Windows (XP ou Vista) avec une architecture processeur de 32bits, d'autres sous Linux (Ubuntu, Debian, SUSE, etc.) avec une architecture processeur de 64 et 32bits, certaines peuvent avoir une faible capacité de mémoire, etc. Le système d'administration doit tenir compte de cette caractéristique d'hétérogénéité lors de l'administration de l'application.

Contributions

Nous présentons dans cette section un aperçu de nos différentes propositions pour palier aux problèmes précédemment présentés : ***expressivité***, ***performance*** et ***hétérogénéité***.

La première contribution porte sur l'***expressivité*** du formalise de description des paramètres de configuration et de l'architecture de l'application. L'objectif ici est de proposer un formalisme adéquat tenant compte du facteur d'échelle. Pour cela, nous proposons une approche de *description en intension* basée sur un formalisme graphique. Cette approche permet la configuration de nombreuses instances d'entités logicielles et de l'architecture globale de l'application avec beaucoup plus de facilité et moins de verbosité. Ainsi un administrateur n'a plus besoin de décrire chaque instance logicielle à administrer et l'interaction entre chacune de ces instances. Il suffit qu'il décrive de façon intensionnelle l'architecture de son application. Pour une application de type *client/serveur*, l'administrateur peut décrire de façon intensionnelle la configuration de 500 *clients* et 50 *serveurs* dont 10 *clients* par serveur donc 1 *serveur* est interconnecté avec 10 *clients*. Contrairement à l'approche de description en extension qui nécessite pour cet exemple, 500 configuration pour les *clients* et 50 pour les *serveurs*, notre approche ne nécessite qu'une configuration des *clients* et une configuration des *serveurs*.

La deuxième contribution porte sur la ***performance*** de l'administration. L'idée de base consiste à répartir le coût et la charge de l'administration sur plusieurs machines afin d'améliorer la performance en terme de consommation de ressources. L'un des objectifs de cette répartition est de diminuer le temps de déploiement et le temps de réaction (détection de panne, reconfiguration, etc.) permettant ainsi d'augmenter la performance du système d'administration. On parle alors de la **décentralisation de l'administration**. Nous proposons également la **personnalisation de la phase d'installation**. En effet le coût du déploiement est principalement dû à l'installation de l'application sur les machines distantes. Personnaliser cette phase permet de diminuer le coût de l'installation. Par exemple l'administrateur peut mettre des stratégies d'installation en place liées à l'environnement de déploiement : la proximité des clusters, l'installation d'un serveur SAMBA/NFS sur un site, etc.

Enfin la dernière contribution apporte une solution au problème ***d'hétérogénéité***. Cette contribution est scindée en deux parties. La première partie permet de **décrire l'infrastructure matérielle** afin d'exprimer sa topologie, ses caracté-

ristiques et sa structure hétérogène. Dans la deuxième partie, un **gestionnaire de l'hétérogénéité** utilise cette description pour sélectionner et installer le paquetage compatible pour chaque machine selon les caractéristiques matérielles et logicielles.

Dans la suite de cette partie, nous présentons plus en détail dans trois chapitres les contributions précédemment résumées. Pour chaque chapitre, nous commençons par un bref rappel des problèmes auxquels la contribution tente de répondre. Ensuite une description générale de la contribution est présentée. Le contexte de nos travaux porte essentiellement sur le système TUNe. Ainsi nous montrons comment ces contributions sont appliquées puis implantées dans le système TUNe.

Chapitre 5

Expressivité : *Description de l'infrastructure logicielle*

Table des matières

5.1 Rappel du problème

L'administration d'une application nécessite un langage de description des entités logicielles et de l'architecture logicielle de l'application. Dans un contexte de grande échelle, le nombre d'entités logicielle est multiple, il est donc plus commode d'utiliser un langage tenant compte de ce facteur d'échelle. Les langages de description d'architecture logicielle ont pour objectif de fournir une vue structurée de l'application dans l'optique de l'administrer sur une infrastructure matérielle. Ils sont basés sur des concepts communément acceptés : les composants définissent les entités de base d'une application ; les connecteurs définissent les types d'interaction entre les composants ; la configuration définit le paramétrage des composants. La multiplicité d'instances logicielles dans un contexte de grande échelle engendre un problème lors de la description de l'application. Dans la suite de cette section, nous évoquons ces problèmes dans les systèmes : **TUNe**, **JADE** et **GoDIET**.

TUNe : Le système d'administration TUNe propose l'utilisation du formalise UML pour décrire l'architecture et les paramètres de configuration d'une application. Cette description est effectuée dans un diagramme dénommé *diagramme de configuration*. Chaque entité logicielle est décrite dans un élément UML. Les pa-

ramètres de configuration sont représentés par les attributs. L'administration d'une architecture DIET de 10 SeD nécessite 10 éléments UML dans le diagramme de configuration. Cette approche ne passe pas à l'échelle car chaque instance logicielle doit être décrite. On se retrouve dans le *diagramme de configuration* autant d'éléments UML que d'instances logicielles à administrer.

JADE : Le framework JADE utilise le langage Fractal ADL pour décrire l'architecture logicielle et la configuration d'une application. Fractal ADL est un langage déclaratif qui permet de décrire des architectures logicielles à base de composants Fractal. La syntaxe du langage est basée sur XML. L'administration avec JADE nécessite la description de chaque instance logicielle sous forme d'un composant primitif Fractal. Les paramètres de configuration sont représentés par des attributs composants. Les dépendances entre les instances logicielles sont exprimées par des liaisons Fractal. Pour déployer une application DIET de 5 LA et 100 SeD dont chaque LA est relié à 20 SeD, chaque instance SeD, LA doit être décrite (une description de 105 composants Fractal). Ensuite, il faut exprimer la liaison de chaque instance LA aux 20 SeD correspondants. Nous nous retrouvons à la fin de cette opération avec 100 lignes de code XML pour juste effectuer les liaisons entre les LA et les SeD.

GoDIET : Le langage de description utilisé par ce système de déploiement de DIET est également basé sur le formalisme XML. Pour déployer une architecture DIET, chaque instance d'agents (MA et LA) et chaque instance de serveurs SeD est décrite dans un fichier XML propre à GoDIET. Une composition de balise XML est utilisée pour décrire les dépendances entre les agents. Par exemple, pour exprimer qu'un agent MA est lié (interprétation du fils dans la structure arborescente de DIET) à un LA, il faut exprimer dans la description de l'agent MA, celle du LA. Cette approche est à la fois verbeuse et ne permet pas le passage à l'échelle vu que chaque agent ou de serveur doit explicitement être décrit.

L'administration d'une application passe par une phase de description des paramètres de configuration et de l'architecture logicielle. Cette phase a pour but d'exprimer dans un formalisme, l'architecture logicielle et la description des paramètres de configuration de l'application. Cela peut engendrer de problèmes d'expressivités lorsque le nombre d'instances logicielles devient important. Dans ce cas, il faut décrire les paramètres de chaque instance ainsi que la connectivité sous forme de dépendances entre les instances d'entités logicielles. Cette approche que nous appelons *expression en extension* ne passe à l'échelle vu le nombre d'instances logicielles à administrer.

Dans la suite de ce chapitre, nous présentons la contribution apportée au niveau de l'expressivité particulièrement sur la description de l'architecture logicielle et les paramètres de configuration d'une application. Nous commençons par introduire, dans la première section, le principe général de la contribution. Puis nous appliquons cette contribution au système TUNe. Nous terminons par l'implantation d' un algorithme appelé *algorithme de liaison* qui va permettre la construction de pattern d'architecture logicielle.

5.2 Expression en intension et pattern d'architecture

5.2.1 Principe général

L'objectif principal de cette contribution sur ***l'expressivité*** est de proposer un formalisme permettant de décrire plusieurs instances logicielles tout en tenant compte du facteur d'échelle. Le formalisme proposé doit être moins verbeux et permettre la description de nombreuses instances logicielles. Nous proposons pour cela : la **description en intension**. Le concept de **description en intension** permet de décrire les entités logicielles par classe sans décrire chaque instance de cette classe.

Pour une application DIET, on peut décrire une classe MA composée de 5 instances, une classe de 100 LA, une classe de SeD composée de 500 instances serveurs. Il suffit juste de décrire les propriétés de la classe comme paramètres de configuration communs aux instances de cette classe. Au lieu d'effectuer une description de plus 600 instances logicielles (5 MA + 100 LA + 500 SeD), l'administrateur décrit seulement 3 classes (la classe des MA, LA et SeD) d'entités logicielles.

Question : *comment exprimer la dépendance entres les instances des classes ?* Par exemple comment exprimer le fait que :
- chaque instance de MA est reliée à 20 LA (ou à au moins/maximum 20 LA) ;
- chaque instance LA est reliée à une instance de MA ;
- chaque instance LA est reliée à 5 SeD (ou à au moins/maximum 5 SeD) ;
- chaque instance de SeD est reliée à une seule instance de LA.

Pour répondre à cette question, nous proposons la construction de pattern d'architecture. Après la description en intension de chaque classe logicielle, les dépendances entre les instances des classes sont également exprimées. Un algorithme que nous détaillons dans la partie implantation va construire l'architecture globale en extension à partir de la description en intension des classes et l'expression de leurs dépendances. Un exemple d'expression de pattern d'architecture : **description de 50 LA et 100 SeD dont chaque instance LA est reliée à 2 SeD et chaque instance SeD est reliée à une instance LA**.

5.2.2 Application à TUNe

La description de l'architecture d'une application peut être effectuée en utilisant des langages textuels (ADL) mais aussi des langages graphiques. Ainsi un certain nombre de travaux s'orientent sur l'utilisation de la syntaxe et du vocabulaire d'UML pour la conception non seulement des applications, mais également des architectures logicielles dans l'optique de les déployer. Il s'agit donc d'utiliser UML comme un

ADL. En effet la notation UML, depuis sa nouvelle version 2.0, introduit la notion de composants et de diagrammes structurels facilitant la modélisation d'architectures. L'approche dans TUNe consiste à utiliser les concepts UML pour décrire les différentes configurations et l'architecture de l'application. Ce langage graphique et intuitif facilite la description de tout type d'application.

Un profil UML est introduit pour décrire graphiquement l'architecture logicielle et les paramètres de configuration de l'application à administrer. Les entités logicielles sont décrites dans des éléments UML. Les dépendances sont exprimées par la notion d'association. Le premier avantage est que UML est plus intuitif que le langage de description d'architecture basé sur XML et ne nécessite pas un apprentissage supplémentaire aux administrateurs. Un exemple de description d'une architecture DIET est présenté sur la figure 5.1. Cet exemple est composé d'un MA, de 3 LA et 5 SeD dont chaque LA est relié au minimum à 1 SeD et au maximum à 2 SeD. Tous les SeD sont reliés à 1 seul LA. Tous les LA sont reliés au MA. Pour des raisons de lisibilité, nous avons omis les paramètres de configuration.

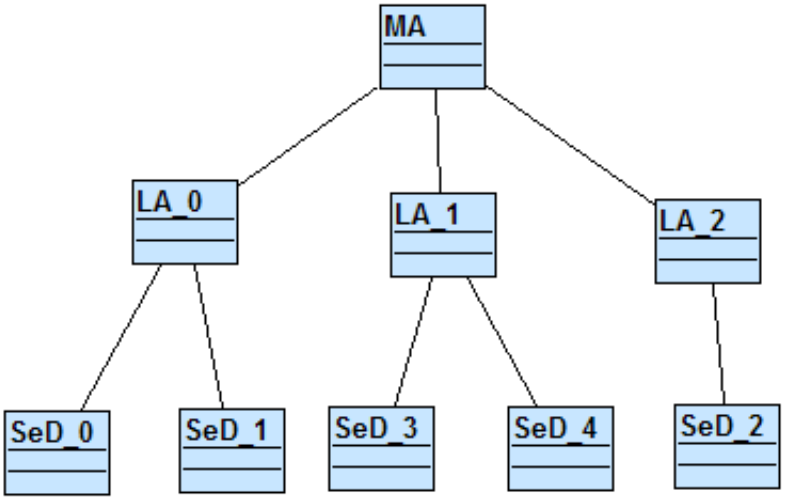

FIGURE 5.1 – Description d'une architecture DIET

Nous proposons d'utiliser des classes UML pour décrire les instances d'une entité logicielle de façon intensionnelle. Cette notion de classe permet la description des instances logicielles qu'on veut déployer. Pour cela, nous introduisons un attribut spécifique **initial** qui indique le nombre initial d'instances logicielles à administrer. Ainsi chaque classe du profil UML correspond à un type d'entités logicielles qui peut être instancié en plusieurs réplicas selon la valeur de l'attribut **initial**. Un exemple de description en intension est illustré par la figure 5.2. Cet exemple montre la description de 500 serveurs SeD d'une application DIET avec illustration de génération du SR équivalent.

Une dépendance entre deux classes d'entités logicielles est représentée par une association au sens UML. Cette association dispose des *ports* (ou *cardinalités*) qui représentent des contraintes sur la dépendance entre les instances de deux classes

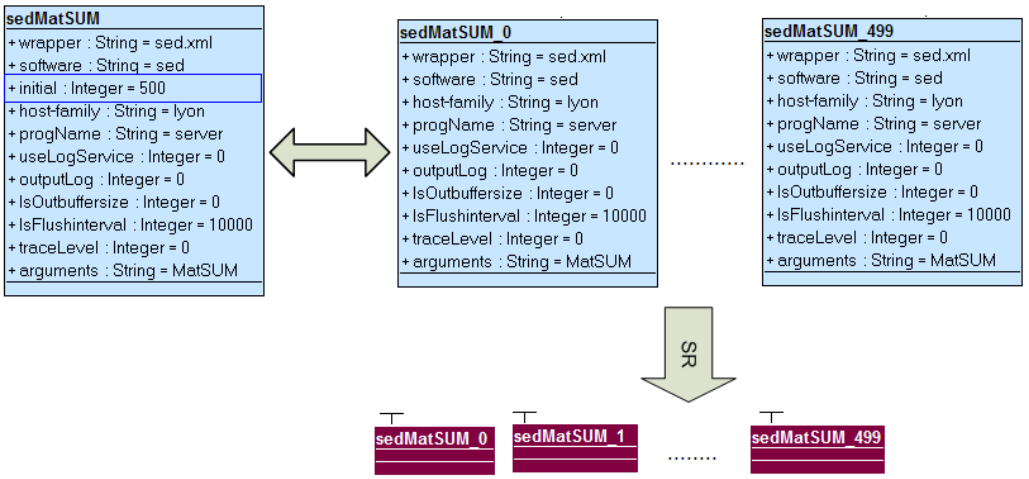

FIGURE 5.2 – Description en intension de 500 serveurs SeD (**sedMatSUM** des serveurs qui effectuent de sommes matricielles)

associées. Un lien entre deux instances génère une liaison entre les composants wrappers Fractal instanciés depuis ces deux instances. Un exemple est présenté par la figure 5.3 où une architecture DIET composée d'une instance MA, de 3 instances LA et de 5 instances SeD. L'instance MA doit être reliée à toutes les instances (*******) LA et chaque instance LA doit être reliée à une seule instance (***1***) de MA. Chaque instance LA doit être reliée à au minimum, une instance et au maximum 2 instances (***1..2***) de SeD. Chaque instance SeD doit être reliée à une seule instance (***1***) LA.

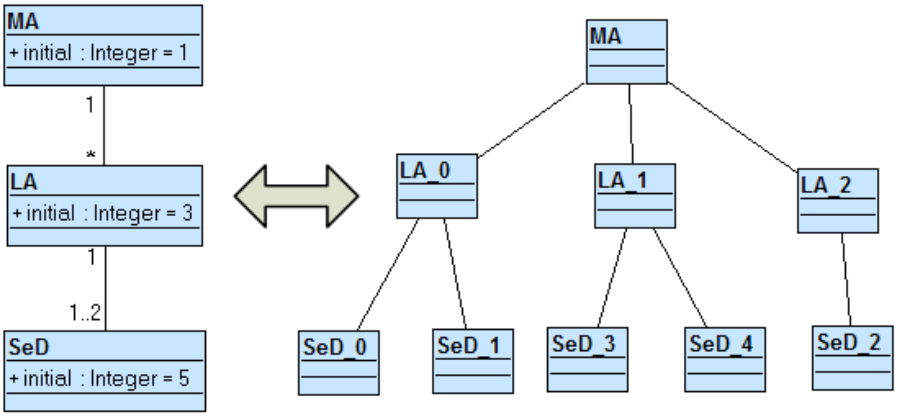

FIGURE 5.3 – Utilisation de ports pour exprimer la dépendance entre les instances de classe

5.2.3 Mise en œuvre dans le système TUNe

Comme évoqué précédemment, on peut remarquer qu'une cardinalité est associée à chaque liaison (***association***). Nous proposons dans cette section la mise en

œuvre d'un algorithme de construction d'architecture logicielle en extension à partir de la description en intension. L'algorithme prend en entrée la description en intension avec les contraintes de cardinalité sur les associations. Nous commençons par expliquer la sémantique des cardinalité sur les associations. Selon le schéma de la figure 5.4, soient **A (initA)** et **B (initB)** deux classes d'entités logicielles reliées où **initA** est la valeur de l'attribut **initial** de la classe du logiciel **A** et **initB** celle du logiciel **B**.

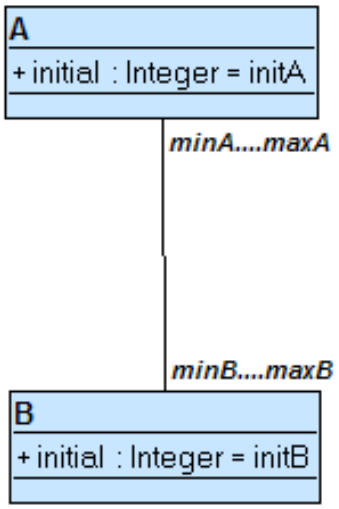

FIGURE 5.4 – Schéma expliquant la sémantique des cardinalités

La sémantique est alors la suivante : **A (initA) minA..maxA—minB..maxB B (initB)** : chaque instance du logiciel **A** doit être reliée au minimum à **minB** instances du logiciel **B** et au plus à **maxB** instances du logiciel **B**. Chaque instance du logiciel **B** doit être reliée au minimum à **minA** instances du logiciel **A** et au maximum **maxA** instances du logiciel **A**. Ainsi il doit y avoir au minimum **minA**×**initB** instances de **A (initA** $\geq$ **minA**×**initB)** et **minB**×**initA** instances de **B (initB** $\geq$ **minB**×**initA)**

Quelques exemples de valeurs de *minA, maxA, minB, maxB* :

- *A (initA) 1—1 B (initB)* : chaque instance du logiciel **A** doit être reliée à une instance du logiciel *B* et chaque instance du logiciel *B* à une instance du logiciel *A*. Ainsi, il doit y avoir autant de logiciels *A* que de logiciels *B* et donc $initB = initA$. Cette cardinalité est par exemple utilisée pour associer une sonde à chaque logiciel. Dans ce cas particulier $minA = maxA = 1$ et $minB = maxB = 1$;
- *A (initA) 1—u B (initB)* : chaque instance du logiciel *A* doit être reliée à *u* instances du logiciel *B* et chaque instance du logiciel *B* doit être reliée à une seule instance du logiciel *A*. Ainsi, nous devons avoir $initA \times u$ logiciels *B* et $initB = initA \times u$. Cette cardinalité est par exemple utilisée pour faire des déploiements en arbre, comme pour DIET.

L'objectif ici est de proposer un algorithme afin d'effectuer les liaisons selon la valeur des cardinalités. L'algorithme doit être capable de construire une architecture logicielle en extension tout en respectant les contraintes sur les cardinalités. Selon le schéma de la figure 5.4, l'exécution de l'algorithme doit lier chaque instance de l'entité **A** au minimum à **minB** de l'entité **B** et au maximum au **maxB** de l'entité **B**. Cette même contrainte doit être vérifiée pour l'entité **B**. Nous proposons un algorithme appelé ***algorithme de liaison*** permettant la génération de l'architecture à composants et d'effectuer les liaisons de façon cohérente entre ces composants logiciels. A partir de la figure 5.4, nous décrivons le principe général de l'algorithme. L'idée principale de l'algorithme est de supposer au préalable que toutes les instances de deux entités logicielles sont liées les unes aux autres. Ensuite les liaisons sont défaites (cassées) jusqu'à ce que les contraintes sur les associations soient satisfaites. La figure 5.5 illustre nos propos.

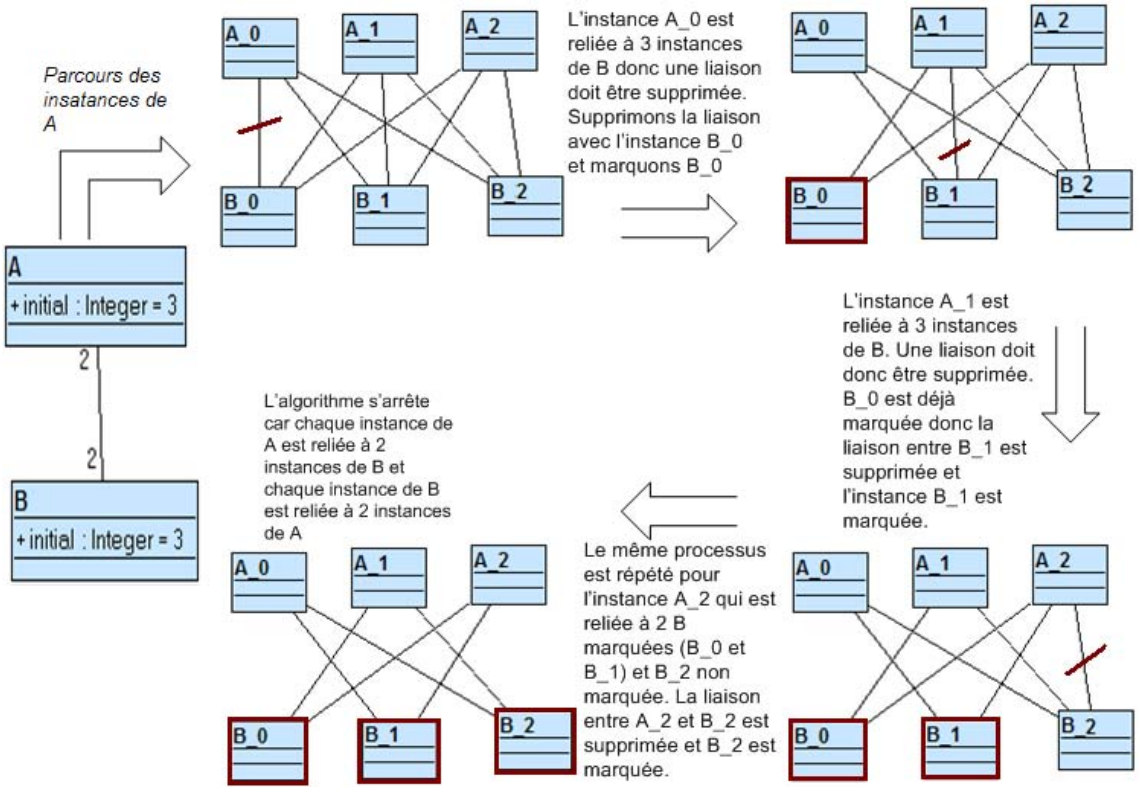

FIGURE 5.5 – Exécution de l'*algorithme de liaison* sur une exemple simple

1. **initA** représente le nombre d'instances de l'entité logicielle **A** (respectivement **initB** pour l'entité **B**) qui doit initialement être déployé.
2. **minA** est le nombre minimum d'instances de l'entité **A** qui doit être lié à une seule instances de l'entité **B**. **maxA** est le nombre maximum d'instances de l'entité **A** qui doit être lié à une seule instance de l'entité **B**.
3. **minB** est le nombre minimum d'instances de l'entité **B** qui doit être lié à une seule instance de l'entité **A**. **maxB** est le nombre maximum d'instances de l'entité **B** qui doit être lié à une seule instance de l'entité **A**.
4. liaison de chaque instance de l'entité **A** à toutes les instances de l'entité **B**. A la fin de cette opération, toutes les instances sont liées les unes aux autres.
5. Si **maxA** > **maxB**, on fait un parcours des instances de l'entité **A** sinon **B**
6. On suppose que **maxA** > **maxB** donc on parcourt l'ensemble des instances de l'entité **A** :

```
Trouver : bool ;
Tant que (il existe une instance de l'entité A, notée A_t, telle que le nombre
d'instances de B liées à A_t > maxB) faire
    Pour (Toute instance de l'entité A, notée A_i) faire
        Trouver ← faux ;
        Si (nombre d'instances de B liées à A_i > maxB) Alors
            Pour (Toute instance de B liée à A_i, notée B_j) faire
                Si (nombre d'instances de A liées à B_j > maxA  ET
                ¬(B_j marqué)) Alors
                    Marquer l'instance B_j ;
                    défaire la liaison entre A_i et B_j ;
                    Trouver ← vrai ;
                    Quitter la boucle Pour ;
                Fin Si
            Fin Pour
            Si ( ¬(Trouver)) Alors
                Pour (Toute instance de B liée à A_i, notée B_j) faire
                    Si (nombre d'instances de A liées à B_j > minA  ET
                    ¬(B_j marqué)) Alors
                        Marquer l'instance B_j ;
                        défaire la liaison entre A_i et B_j ;
                        Quitter la boucle Pour ;
                    Fin Si
                Fin Pour
            Fin Si
        Fin Si
    Fin Pour
    Démarquer toutes les instances de B ;
Fin Tant que
```

Algorithme 1: *Algorithme de liaison*

La figure 5.6 suivante donne un exemple de patterns d'architecture pour une application DIET.

Synthèse : Nous avons proposé dans cette contribution une approche basée sur la **description en intension**. Avec cette **description intensionnelle**, nous

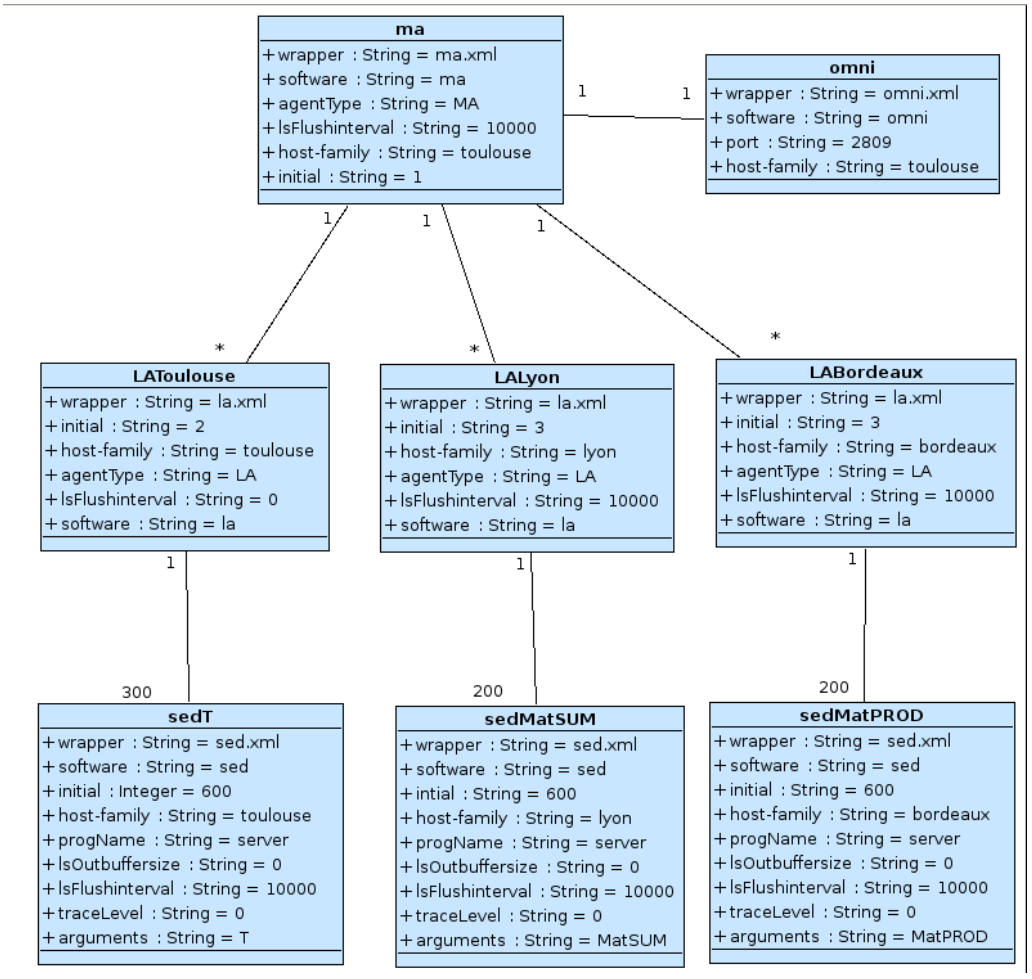

FIGURE 5.6 – Exemple d'architecture logicielle à grande échelle pour une application DIET

pouvons décrire plusieurs instances logicielles avec un minimum de verbosités. La relation de dépendance entre les entités logicielles est représentée par une association. Les cardinalités de l'association représentent la relation entre instances d'entités logicielles. Un ***algorithme de liaison*** est proposé pour construire l'architecture logicielle en extension. Cet algorithme utilise la description en intension et la relation de dépendance entre les entités logicielles pour construire la description en extension correspondante.

Chapitre 6

Performance : *Décentralisation de l'administration et personnalisation de l'installation*

Table des matières

6.1 Rappel du problème

Comme l'a mis en évidence le chapitre 2, les grilles de calculs sont constituées d'un ensemble de clusters situés géographiquement sur des sites différents. Un cluster est composé d'un ensemble de machines homogènes. Nous nous intéressons plus particulièrement ici au coût que peut engendrer le processus d'administration sur la machine qui administre (*la machine d'administration*). Ce coût va dépendre à la fois du coût de la communication entre les machines et la consommation de ressources par la *machine d'administration*. Il existe plusieurs ordre de grandeur pour le coût de la communication :

- entre deux threads d'un processus ou entre deux processus d'une même machine ;
- entre deux machines reliées par un réseau haute performance d'un cluster (SAN) ;
- entre deux clusters dans un même réseau local (LAN) ;
- entre deux sites d'un même état (WAN continental) ;
- et entre deux continents distincts (WAN intercontinental).

Nous nous intéressons à la connexion entre deux machines pour analyser le coût de la communication. Cette connexion utilise des protocoles de communications pour accéder à une machine. Ces protocoles permettent d'effectuer une opération (exécution d'un programme, copie des fichiers, etc.) sur une machine distante. Les plus utilisés sont *rsh* et *ssh*. Du point de vue technique, ces protocoles utilisent plusieurs échanges lors de l'exécution d'un programme sur la machine distante et présentent donc une charge pour la machine qui administre. Ces échanges consistent en : (i) la création d'un processus sur la machine distante, (ii) l'établissement d'une liaison TCP permettant d'interagir avec le processus distant et (iii) la mise en place éventuelle sur un nœud distant d'un processus de contrôle (redirection des entrées/sorties et des signaux de contrôle du processus distant).

En plus du coût de communication réseau dû à la connexion sur les machines pour l'exécution des tâches d'administration, il y a des ressources consommées par les processus d'administration. En effet le processus d'administration passe par une phase d'installation. Cette phase consiste généralement à exécuter plusieurs opérations notamment les copies des paquetages et des librairies des entités logicielles, configuration de l'application, démarrage des différentes entités logicielles sur les machines distantes, etc. Toutes ces opérations nécessitent la création de plusieurs processus sur la machine qui administre donc une consommation de ressources de cette machine (mémoire, disque, etc.).

Le coût et la consommation de ressources est d'autant plus élevé que l'administration s'effectue dans un contexte de grande échelle et que les tâches d'administration sont exécutées par une seule machine (*administration centralisée*). En effet les ressources d'une machine sont limitées notamment le nombre de connexions distantes autorisées (1024 sous Linux) ou le nombre de processus maximum pour un utilisateur (254 sous Linux). Nous proposons donc une *approche décentralisée* qui consiste à effectuer l'administration à partir de plusieurs machines. Cette approche va permettre de contourner les limites de l'approche centralisée car les coûts et la consommation de ressources sont répartis entre les machines qui administrent permettant ainsi d'augmenter la performance du système d'administration.

Comme évoqué précédemment, le coût de l'administration est dû essentiellement à la communication réseau (entre la machine qui administre et les machines administrées) et l'exécution des opérations d'administration notamment le processus d'installation. Ce processus est caractérisé par un enchaînement de plusieurs opérations, généralement codées *en dur* dans le code source des systèmes d'administration. L'exécution de ces opérations est automatiquement effectuée par le système

d'administration lors de l'installation d'une application. Il est donc impossible de personnaliser cette exécution afin par exemple de maitriser la copie des paquetages ou tenir compte de la spécificité de l'environnement d'exécution (des paquetages déjà installés sur les machines, existence d'un serveur SAMBA/NFS sur une partie de la grille, etc.)

Synthèse :

La première problématique que nous avons abordée dans cette section est l'analyse du coût et la charge de l'administration. En effet l'administration d'une application a un coût marginal à une petite échelle comme sur un cluster et un coût important à une grande échelle comme sur la grille. Ce coût provient essentiellement des différentes tâches du déploiement (installation des paquetages, configuration, démarrage) et de reconfiguration (détection de panne, réparation, etc.). La deuxième problématique présentée porte sur les processus d'installation. Cette phase du déploiement est généralement implantée *en dur* dans le code source des systèmes d'administration. Dans ce cas, les processus d'installation sont automatiquement exécutés sans possibilité de contrôle.

Dans la suite, nous présentons dans une première partie l'approche basée sur la *décentralisation de l'administration* afin de distribuer le coût et la charge. La personnalisation de la phase d'installation est présentée en vue de donner la possibilité aux administrateurs d'une part de tenir compte du contexte et de la spécificité de l'infrastructure matérielle lors de l'administration ; d'autre part de pouvoir mettre en place des stratégies d'installation de leurs applications afin d'augmenter la performance.

6.2 Décentralisation de l'administration

6.2.1 Principe général

Comme nous avons vu dans la section précédente, l'approche centralisée ne passe pas à l'échelle. Dans cette approche, une seule machine est chargée d'effectuer tous les appels distants nécessaires au déploiement et à la gestion de l'application. L'administration devient difficile lorsque l'application est composée de plusieurs milliers d'entités logicielles qui doivent être déployées et gérées sur un ensemble de sites distribués. L'augmentation du nombre de participants (nœuds et entités logicielles) entraîne une augmentation du coût et de la charge de l'administration.

Une solution simple permettant de contourner ces limites est de *décentraliser l'administration* [TH09] [TSHB10]. Cela consiste à répartir l'exécution des tâches d'administration sur plusieurs machines. Ainsi le coût et la charge sont répartis entre plusieurs nœuds. Pour cela nous proposons de déployer plusieurs systèmes d'administration. Chaque système d'administration déployé va se charger de l'ad-

ministration une partie de l'application, donc se voir déléguer une partie des tâches d'administration.

Question : *comment procéder au déploiement des systèmes d'administration ?*

Pour répondre à la question, nous considérons un système d'administration comme une application. Cette application peut donc être déployée par le système d'administration. Autrement dit le système d'administration va s'auto-déployer. L'objectif est de faire participer un système d'administration au déploiement de ses instances.

Question : *comment spécifier les systèmes d'administration à déployer ? Comment déterminer le nombre de systèmes d'administration à déployer ?*

Pour spécifier les systèmes d'administration à déployer, nous proposons, pour des raisons de performances et de souplesses que l'architecture des systèmes d'administration à déployer soit décrite de façon arborescente par l'administrateur. Cela va permettre à un administrateur d'une part de spécifier l'ensemble de systèmes qu'il veut déployer ; d'autre part d'exprimer comment le déploiement de ces systèmes doit être effectué. Chaque nœud de cette arborescence représente un système d'administration à déployer. La sémantique de lien entre père et fils de l'arbre est le déploiement. Chaque nœud de l'arbre va déployer ses fils. Ainsi nous obtenons un déploiement décentralisé et hiérarchique des systèmes d'administration. Le problème qui se pose, est le déploiement de la racine de l'arbre vu que chaque fils est déployé par son père. Pour cela nous proposons que le système racine soit lancé par l'administrateur. Ce système racine est alors responsable de poursuivre le déploiement des autres systèmes d'administration en occurrence ses fils. Une fois que tous les systèmes d'administration sont déployés, l'administration proprement dite de l'application peut débuter.

Question : *comment les tâches d'administration sont déléguées entre les systèmes d'administration ? Ou comment spécifier à un système d'administrer telle partie de l'application ?*

Après le déploiement des systèmes d'administration, ceux-ci sont chargés d'administrer une partie de l'application. Cette information (la partie de l'application qu'un système doit administrer) est déterminée lors de la description des systèmes. En effet, nous proposons de paramétrer les systèmes d'administration. Ainsi plusieurs paramètres de configuration sont introduits afin de spécifier : les parties de l'application à administrer, le nœud ou le cluster sur lequel le système est déployé, etc.

Notre approche vise à répartir les tâches d'administration sur plusieurs nœuds. Le seul système d'administration qui effectue l'administration, dans le cas d'une approche centralisée, est remplacé par un ensemble de systèmes d'administration organisé de façon hiérarchique favorisant l'efficacité de l'administration. Cette approche offre divers avantages :

- une meilleure répartition de la charge entre les différents systèmes d'administration ;
- une plus grande stabilité du système (si un des systèmes d'administration tombe en panne, les autres systèmes d'administration peuvent se réorganiser pour le réparer) ;
- une gestion simplifiée et autonome de chaque système d'administration (par exemple dans le cas d'une grille, on peut déployer un système d'administration par cluster. Chaque système d'administration est responsable des applications déployées sur son cluster).

La figure 6.1 suivante résume le processus d'administration décentralisée.

Selon la figure 6.1, les modifications à effectuer par rapport à une administration centralisée sont : (i) la description de la hiérarchie et les paramètres de configuration des systèmes d'administration ; (ii) le déploiement des systèmes d'administration (le premier système qui est la racine de l'arbre est déployé par l'administrateur) ; (iii) déclenchement du processus d'administration par les systèmes précédemment déployés.

En résumé, nous avons présenté dans cette section la *décentralisation de l'administration* afin de répartir la charge et le coût de l'administration sur plusieurs machines. Les systèmes d'administration sont décrits sous forme arborescente et déployés de façon hiérarchique. Chaque système d'administration (chaque nœud) de l'arbre déploie ses fils (déploiement des fils par le père). La racine de la hiérarchie est déployée par l'administrateur. Les systèmes d'administration sont paramétrés pour par exemples déterminer : les tâches d'administration à effectuer, le nœud ou le cluster d'administration.

6.2.2 Application à TUNe

Dans cette section, nous montrons comment appliquer l'approche de décentralisation au système TUNe. Nous commençons par présenter le déploiement hiérarchique de TUNe. Comme évoqué dans la section précédente, la hiérarchisation nécessite un formalisme permettant de l'exprimer. Nous présentons donc un formalisme pour exprimer cette hiérarchie des TUNe à déployer. Nous présentons également plus en détails la délégation des tâches entre les TUNe. Enfin nous abordons un aperçu sur la mise en œuvre (implantation) dans le système TUNe.

Déploiement hiérarchique des TUNe

L'objectif ici est de déployer plusieurs TUNe de façon hiérarchique. Les TUNe déployés forment un arbre dont la racine est déployée par l'administrateur. Chaque nœud (sauf la racine) de l'arbre est déployé par son père. Chaque TUNe déployé est chargé d'administrer une partie de l'application de façon autonome.

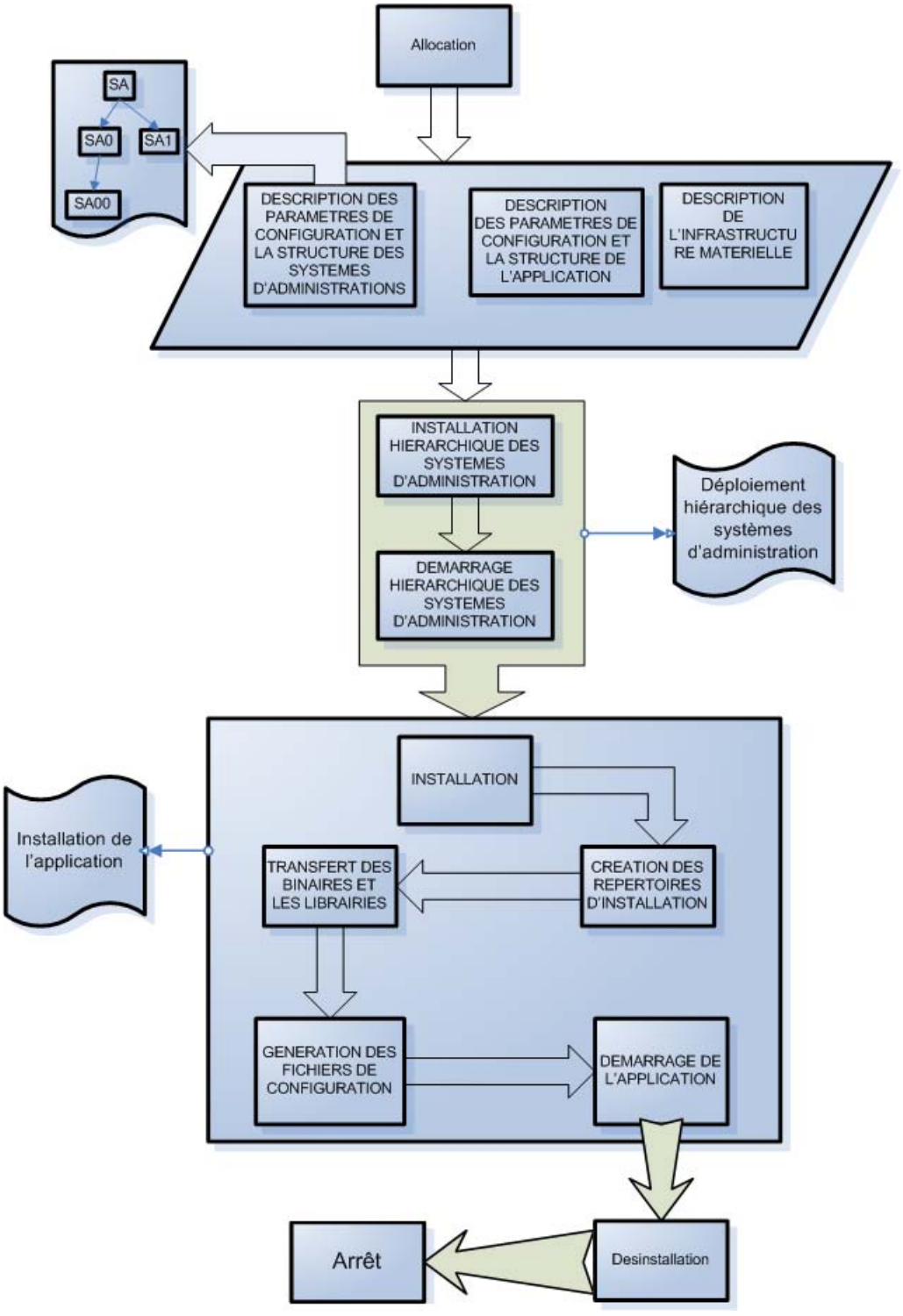

FIGURE 6.1 – Processus d'administration décentralisée pour les applications patrimoniales

Pour déployer une hiérarchie de TUNe, l'administrateur doit décrire la structure et les paramètres de configuration de chaque TUNe de la hiérarchie. Pour cela nous proposons d'introduire un nouveau diagramme pour exprimer cette description. Ce

diagramme que nous appelons ***diagramme d'administration*** est utilisé pour définir l'architecture de la hiérarchie ainsi que les différentes configurations des TUNe à déployer. La figure 6.2 montre l'exemple d'un ***diagramme d'administration***.

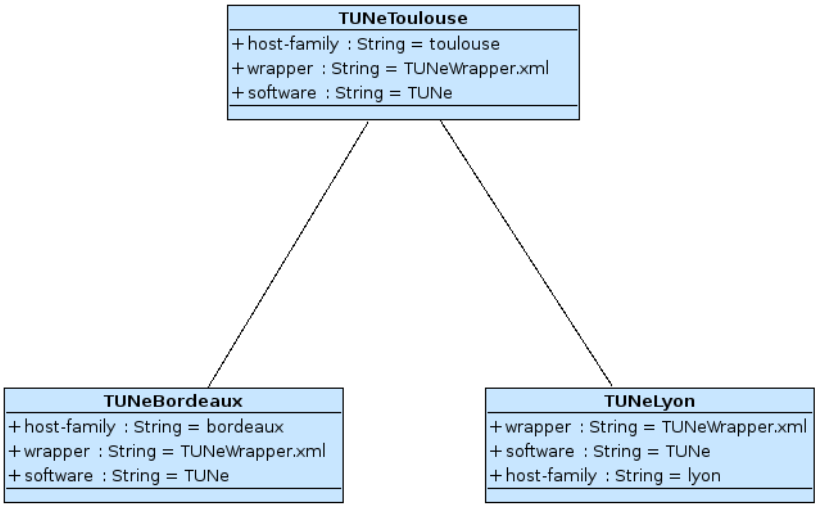

FIGURE 6.2 – Exemple d'un ***diagramme d'administration***

Dans l'exemple de la figure, **TUNeToulouse** est déployé par l'administrateur. **TUNeToulouse** se charge d'interpréter le diagramme afin de connaître ses fils qu'il doit déployer et poursuit le déploiement des TUNe : **TUNeBordeaux** et **TUNeLyon**.

Avec cette hiérarchisation, les tâches d'administration sont réparties entre les TUNe. Chaque TUNe est chargé de l'exécution d'une partie des tâches d'administration. Par ailleurs, les TUNe père et fils peuvent se déléguer de tâches. Cette délégation de tâches est nécessaire lorsqu'un TUNe veut par exemple, effectuer une opération d'administration (exemple redémarrer) sur une instance d'entité logicielle qui n'est pas sous son contrôle. La délégation de tâches entre les TUNe s'effectue par le protocole RMI. Nous reviendrons plus en détail sur ce dernier point dans la partie mise en œuvre.

Question : *Comment déléguer une tâche d'administration à TUNe ? comment exprimer, déterminer qu'un TUNe doit administrer une telle partie de l'application ?*

Délégation de tâches d'administration entre les TUNe

Nous avons vu dans le chapitre 5 (contribution portant sur l'*expressivité*) que chaque entité logicielle est paramétrée par un attribut spécifique **host-family** qui indique le cluster sur lequel ses instances doivent être administrées. La valeur de l'attribut **host-family** est un cluster défini dans le diagramme de nœud. Le **diagramme de nœud** définit les clusters sur lesquels les entités logicielles de l'application vont

être administrées. Les TUNe sont également paramétrés par le même attribut **host-family**. Nous utilisons ces informations pour déléguer les tâches d'administration aux TUNe.

Le paramètre de configuration **host-family** d'un TUNe va définir à la foi le cluster sur lequel il est déployé (donc sur une machine du cluster) et les entités logicielles qu'il administre. Autrement dit, un TUNe administre les entités logicielles de son cluster.

Lors du processus d'administration, une *projection* entre le diagramme qui décrit l'infrastructure matérielle (**diagramme de nœud**) et celui qui décrit les l'architecture de TUNe (**diagramme d'administration**) est effectuée. Cette projection permet d'affecter (en fonction du **host-family**) à chaque TUNe les entités logicielles qu'il administre.

Administration de l'application par les TUNe

L'administration d'une application commence par le déploiement des TUNe. Ensuite chaque TUNe entame l'administration d'une partie de l'application selon la valeur de son (**host-family**). L'administrateur lance TUNe avec la description des différents éléments (*diagramme d'administration, diagramme de configuration, diagramme de description de l'infrastructure matérielle, diagrammes d'état-transition de configuration...*). Ce premier TUNe déployé correspond au TUNe racine qui se charge de déployer ses fils. La figure 6.6 montre l'enchaînement du processus d'administration décentralisée avec les différents diagrammes représentés sur les figures(6.3, 6.4, 6.5)

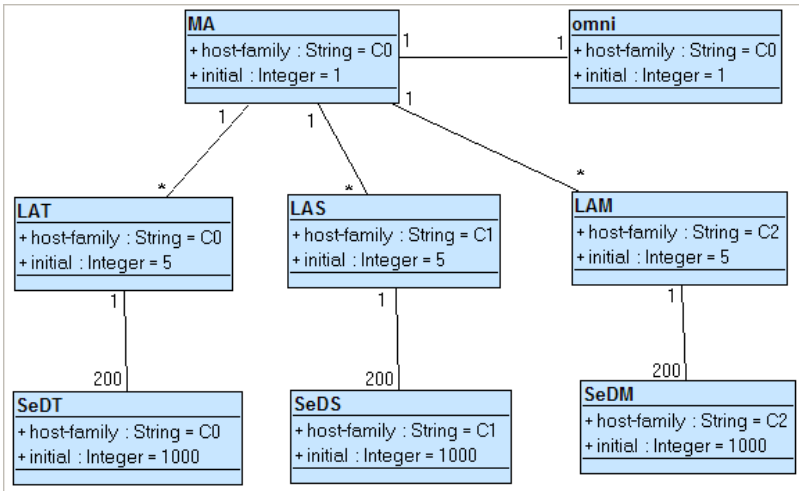

FIGURE 6.3 – Diagramme de configuration

La figure 6.6 montre un exemple d'administration d'une architecture DIET composée de huit entités logicielles (*omni, MA, LAT, LAS, LAM, SeDT, SeDS et SeDM*). L'application est déployée par trois TUNe. Le déploiement commence par

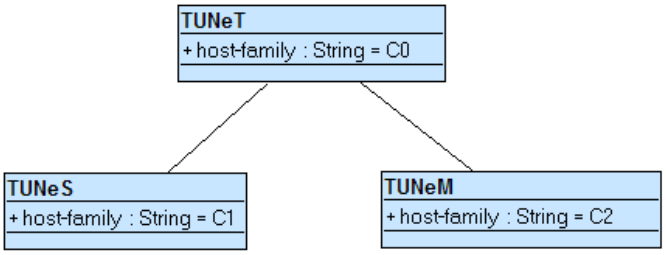

FIGURE 6.4 – Diagramme d'administration

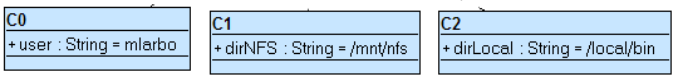

FIGURE 6.5 – Diagramme de grid

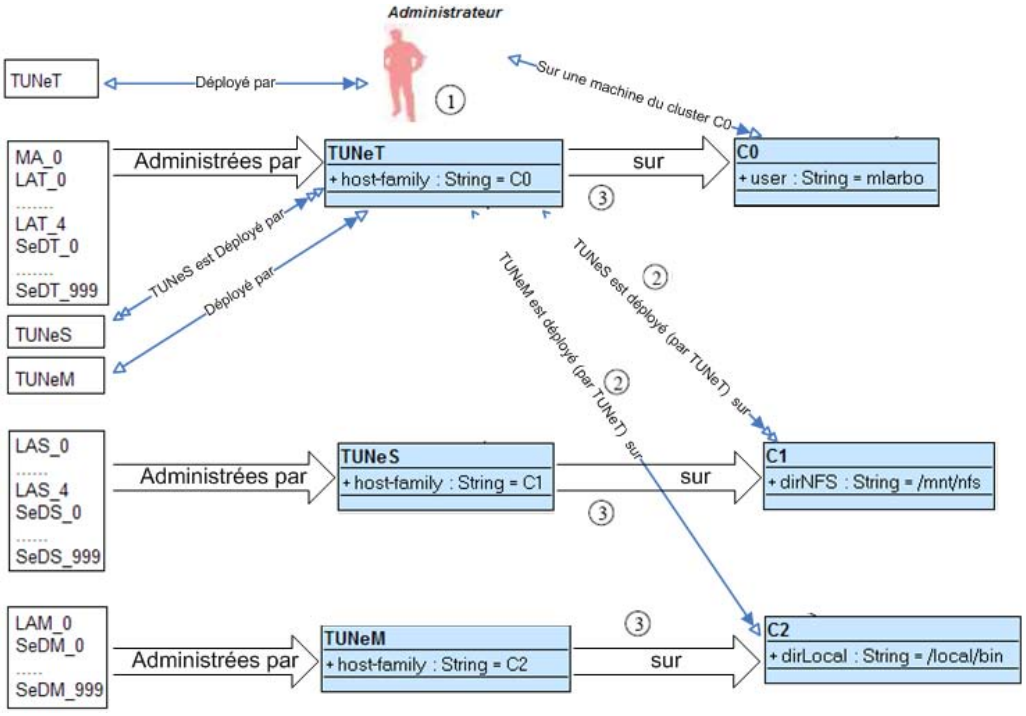

FIGURE 6.6 – Processus d'administration décentralisée

celui de TUNe racine (*TUNeT*). Ce TUNe est déployé par l'administrateur sur le cluster *C0* (*étape 1*). *TUNeT* se charge à son tour de déployer les deux TUNe *TUNeS* et *TUNeM* (*étape 2*). Après le déploiement de la hiérarchie des TUNe, chaque TUNe entame le processus de déploiement de la partie de l'application qui lui est déléguée. Ainsi *TUNeT* déploie les entités de son *host-family* (toutes les instances d'*omni, MA, LAT et SeDT*), *TUNeS* déploie (*LAS, SeDS*) et *TUNeM* déploye les instances

de l'entité *LAM* et *SeDM* (*étape 3*). Ce processus se termine par le démarrage des différentes instances de l'application sur les nœuds physiques.

Après le déploiement des TUNe et l'installation de la partie de l'application déléguée à chaque TUNe, le problème de synchronisation du démarrage des différentes entités logicielles de l'application s'impose. Pour clarifier nos propos, nous prenons un exemple sur une architecture DIET composée d'une instance *MA*, d'une instance LA (*LA1*) reliée au *MA* et de deux types SeD de plusieurs instances (*SeD1* de n instances, *SeD2* de m instances). Supposons que les tâches d'administration du *MA*, *SeD1* soient déléguées à un TUNe *T1* et *LA1*, *SeD2* au TUNe *T2*. Pour une architecture DIET le *MA* doit être démarré avant les *LA* qui doivent à leur tour être démarrés avant les *SeD* donc *T2* ne peut pas démarrer les entités logicielles (*LA1, SeD2*) sans que *TI* ait démarré le *MA*. Il faut respecter l'ordre de démarrage global de l'application et synchroniser ce démarrage. Dans notre exemple, une solution consiste à faire démarrer le *MA* par TUNe *T1*. Ensuite *T1* effectue une demande de démarrage de *LA1* à *T2* et attend que ce dernier termine ce démarrage. *T2* reçoit cette demande et entame l'exécution du processus de démarrage du *LA1*. Après le démarrage du *LA1* par *T2* et un retour de confirmation de terminaison, *T1* peut démarrer toutes les n instances du serveur *SeD1* puis effectue à nouveau une demande de démarrage des m instances du serveur *SeD2* à TUNe *T2*. Nous constatons dans cet exemple que le processus de démarrage est orchestré par un TUNe (*T1*). Dans le cas du système TUNe, le processus de démarrage est définit dans le diagramme d'état-transition **startchart**. Nous proposons donc de faire exécuter ce diagramme par TUNe racine de la hiérarchie. Ce dernier va se charger de la synchronisation en effectuant des appels aux autres TUNe.

Nous avons donc délégué la tâche d'exécution du diagramme de démarrage au TUNe racine. Cependant tout appel de méthode (par exemple le démarrage d'une entité logicielle) à partir du diagramme **startchart** sur une entité non administrée par TUNe racine est transmis au TUNe ayant la charge d'administrer cette entité. Cet appel de méthode précise le nom de l'entité concernée et le nom de la provenance de l'appel (nom du TUNe). Lors de la réception d'un appel par un TUNe, celui-ci tente d'exécuter la méthode sur toutes les instances de l'entité. En cas d'échec, l'appel de méthode est soit transmis à ses fils, après avoir vérifié l'existence d'un fils qui administre l'entité concernée, soit transmis au père si l'appel ne provient pas de ce dernier. Cela permet de limiter la transmission d'appel de méthodes entre les TUNe. La figure 6.8 suivante résume le processus d'*administration décentralisée* avec le diagramme de démarrage représenté sur la figure 6.7.

Pour illustrer nos propos, nous expliquons l'exécution du diagramme **startchart** (figure 6.8) correspondant à l'application de la figure 6.6. Ce diagramme est exécuté par **TUNeT** (TUNe racine). L'exécution commence par le démarrage des entités *omni* et *MA*. Cela consiste à exécuter *omni.start* suivi de *MA.start*. Cette séquence d'exécution déclenche le démarrage des instances *d'omni* et *MA* en effectuant les appels de méthode *start* sur ces dernières. Les deux entités *omni* et *MA* sont sous l'administration de **TUNeT** (de **host-family C0**), par conséquence les

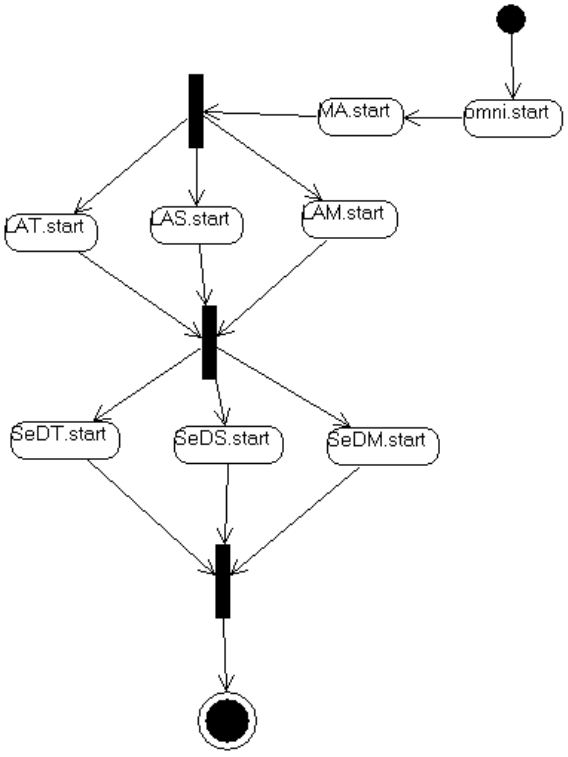

FIGURE 6.7 – Le diagramme de démarrage *startchart* de l'architecture DIET de la figure **??**

exécutions précédentes sont effectuées par **TUNeT**. **TUNeT** poursuit l'exécution du diagramme par le démarrage des *LA* (*LAT.start, LAS.start, LAM.start*) de façon parallèle. Il exécute l'appel de méthode *start* sur les instances de *LAT* sous son administration. Cependant les deux autres appels (*LAS.start, LAM.start*), sont transmis aux fils. **TUNeT** envoie *LAS.start* au **TUNeS** et *LAM.start* au **TUNeM**. Lors de la réception de cet appel par les deux TUNe, chacun exécute la méthode *start* sur les instances de l'entité indiquée dans l'appel. Une attente de synchronisation est effectuée afin que le démarrage des LA soit terminé avant d'entamer celui des SeD. La suite de l'exécution est le démarrage des SeD (*SeDT.start, SeDS.start, SeDM.start*). **TUNeT** exécute *SeDT.start* et transmet les appels de méthodes *SeDS.start et SeDM.start* à ses fils ayant la charge de l'administration de *SeDS* et *SeDM*. Ces derniers sont chargés d'effectuer le démarrage des serveurs(*SeDS, SeDM*) de façon locale.

En résumé, le processus d'administration décentralisée s'effectue comme suit :
- L'administrateur déploie le premier TUNe (la racine de la hiérarchie) ;
- Chaque TUNe de la hiérarchie déploie ses fils ;
- Les TUNe déployés entament le déploiement de l'application proprement dit ;
- Enfin le démarrage de l'application est effectué par TUNe racine qui se charge de l'exécution du diagramme d'état-transition (**startchart**) et la délégation

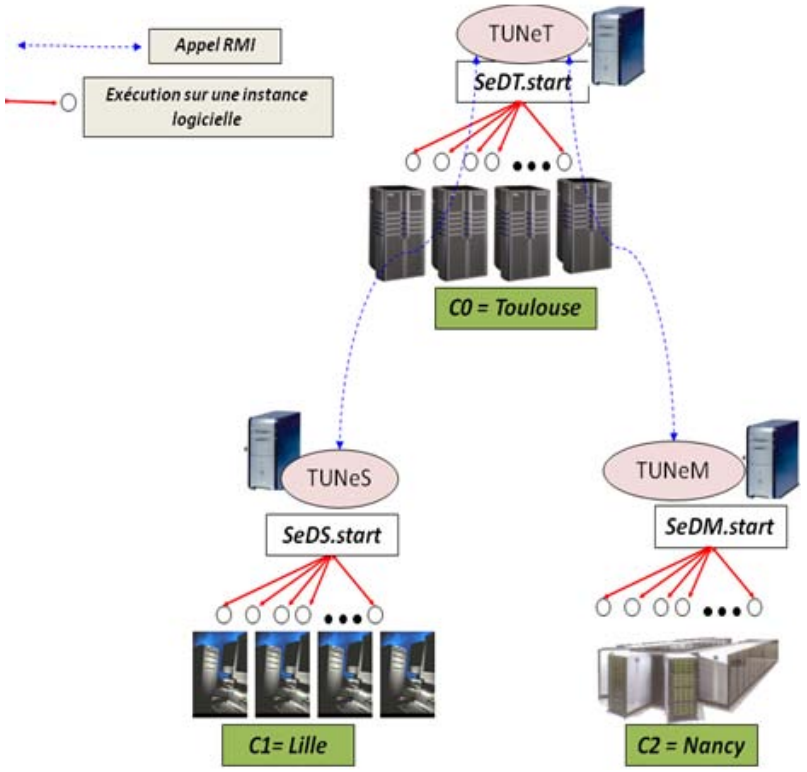

FIGURE 6.8 – Délégation de la tâche de démarrage des instances logicielles d'une application

des tâches de démarrage de chaque entité logicielle aux TUNe appropriés (chargé de l'administration de l'entité logicielle).

6.2.3 Mise en œuvre dans le système TUNe

Nous avons vu dans le chapitre 3 sur la présentation de TUNe qu'il y a deux niveaux d'exécution. Le premier est le niveau du *Systèm Representation* (SR) et le second est le niveau *patrimonial*. Nous présentons dans cette section les modifications effectuées sur le code source de TUNe afin de répartir le niveau patrimonial et sauvegarder le niveau SR. Nous présentons ensuite comment les TUNe communiquent afin de déléguer les tâches d'administration (demande de démarrage d'une entité logicielle, reconfiguration d'une instance d'entité logicielle etc.)

6.2.3.1 Répartition des niveaux d'exécution : SR et patrimonial

Comme évoqué précédemment, TUNe a deux niveaux d'exécution (figure 3.3 du chapitre 3 contexte applicatif). Le niveau *System Representation SR* qui est la re-

présentation sous forme des composants Fractal de l'architecture de l'application. Ce niveau s'exécute sur la machine de TUNe (la *machine d'administration*). Le second est le niveau *patrimonial* qui est la représentation du SR sur les machines physiques donc réparti sur plusieurs machines. Un composant wrapper d'une instance logicielle du niveau SR permet d'orchestrer le processus d'administration de cette instance logicielle au niveau patrimonial. Par exemple, pour une architecture DIET composée d'une instance MA reliée à une instance LA, TUNe crée deux composants wrapper Fractal reliés par une liaison Fractal qui se chargent du processus d'administration. Cette architecture Fractal correspond au SR. Si on effectue par exemple un appel du type *MA.start*, le composant wrapper Fractal va se charger de démarrer l'entité MA sur la machine physique allouée au MA.

Question : *Dans le cas de plusieurs TUNe, faut-il dupliquer ou fragmenter le SR au niveau des TUNe ?*

Nous présentons dans la suite, comment les deux niveaux (SR et patrimonial) sont répartis dans l'extension de TUNe (décentralisé).

Introduction des wrappers proxy et réels

Pour distinguer les wrappers des entités logicielles administrées par un TUNe et ceux qui ne les sont pas, nous avons introduit la notion de composant wrapper *réel* et *proxy*. Dans TUNe hiérarchisé, nous disposons de deux types de composant wrapper au niveau SR à savoir :

- *Composant wrapper réel* : qui représente le composant wrapper d'une entité logicielle administrée par TUNe local ;
- *Composant wrapper proxy* : utilisé comme un représentant d'un composant wrapper réel administré par un autre TUNe (donc pas sous l'administration du TUNe local).

Un composant *wrapper proxy* d'un TUNe a pour fonction de relayer les requêtes de demande de service entre les composants *wrappers réels* de ce TUNe et ceux des autres TUNe. Un service d'un composant *wrapper réel* local à un TUNe (donc administré par TUNe local) peut appeler un service d'un autre composant *wrapper réel* distant (administré par un autre TUNe) comme si celui-ci était fourni par un *wrapper réel* local (i.e. l'interface d'administration n'est pas modifiée). Le *wrapper proxy* fournit les mêmes services que le *wrapper réel*. Néanmoins, les services fonctionnels ne sont pas exécutés sur ce wrapper. Chaque composant *wrapper réel* admet au sein du SR d'un autre TUNe le composant *wrapper proxy* qui le présente. Ainsi, lorsqu'un composant *wrapper réel* d'une entité logicielle demande un service à un composant *wrapper proxy*, cette demande est transmise par TUNe local au TUNe ayant la charge d'administrer cette entité logicielle. Le résultat est ensuite récupéré par le TUNe distant et transmis au TUNe local. Dans l'exemple de la figure 6.9, le composant wrapper du *LA1* de *TUNe1* peut lire un paramètre de configuration du composant *MA* (par exemple le paramètre *$this.MA.nodeName* : le nom de la ma-

chine d'administration du *MA*, *this* pour désigner le composant *wrapper réel LA1*) en passant par le *wrapper proxy* du *MA*. Ce composant proxy s'adresse à *TUNe1* qui se charge de transmettre la requête de lecture à *TUNe0*. *TUNe0* récupère le paramètre de configuration (*nodeName*) du composant *wrapper réel MA* et transmet le résultat à *TUNe1*.

Au niveau patrimonial, seuls les composants wrappers réels sont représentés. En effet les wrappers proxy ne sont pas administrés, il n'est donc pas nécessaire qu'ils soient représentés au niveau patrimonial. La figure 6.9 illustre nos propos. Pour *TUNe0*, seules les entités logicielles *MA*, *LA2* et *SeD2* qu'il administre sont représentées au niveau patrimonial. *TUNe1* représente au niveau patrimonial les entités *LA1* et *SeD1* qui sont sous son contrôle.

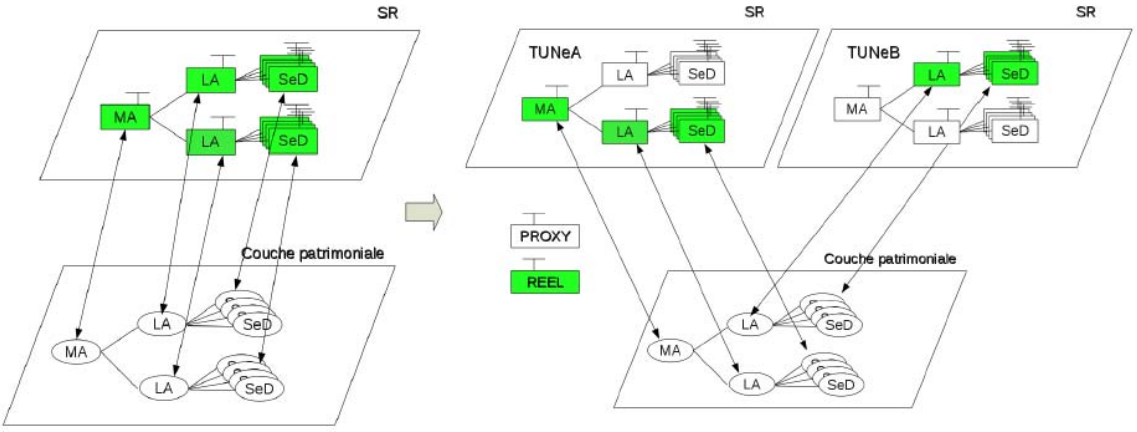

FIGURE 6.9 – Répartition des niveaux

Nous avons vu que la communication entre les composants wrappers réels et proxy passent la communication entre les TUNe.

Question : *comment les TUNe communiquent ?*

Communication entre les TUNe

Pour qu'un wrapper proxy d'un TUNe local demande un service (par exemple la lecture d'un paramètre de configuration) avec un wrapper réel d'un TUNe distant, le wrapper proxy passe par TUNe local. TUNe local se charge de transmettre la demande (de lecture) au TUNe disant. Pour établir cette transmission, il est nécessaire de faire communiquer les TUNe. La communication entre les TUNe est basée sur le protocole RMI. En effet chaque TUNe dispose d'un serveur éventuellement de plusieurs clients RMI permettant la communication *inter-TUNe* (entre les TUNe). L'interface de ce serveur RMI permet d'accéder aux *composants wrappers réels* d'un TUNe (accéder aux paramètres de configuration, exécuter une méthode sur une entité logicielle, etc.). La figure 6.10 suivante illustre nos propos.

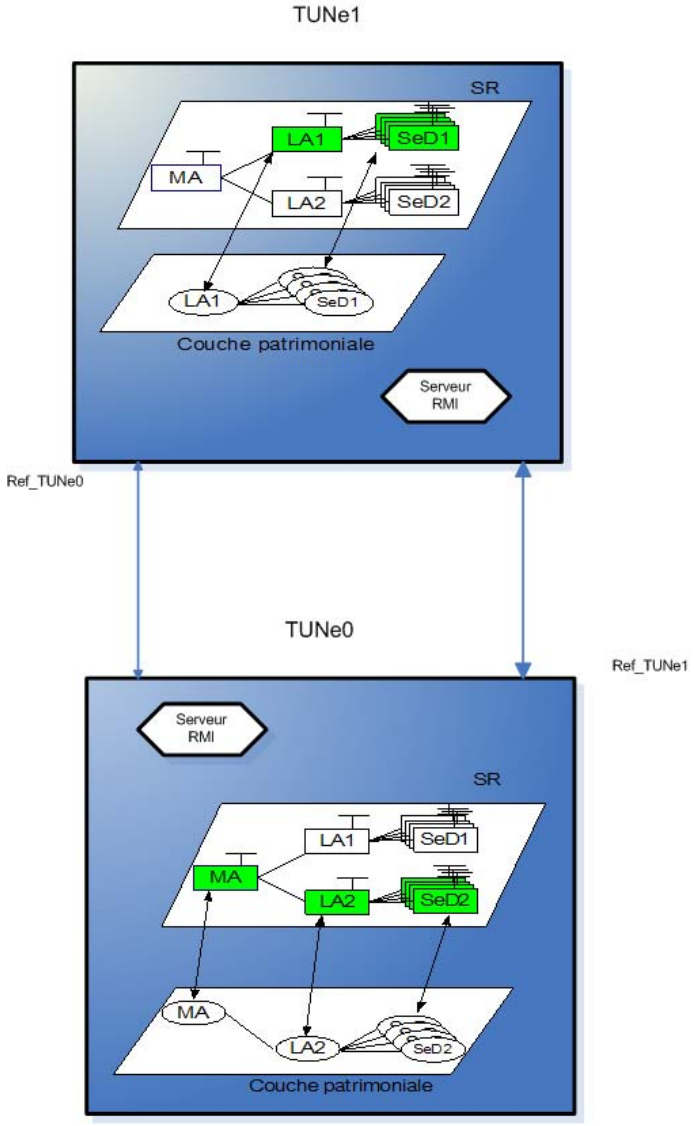

FIGURE 6.10 – Communication entre les TUNe

L'architecture DIET de la figure 6.10 est administrée par deux TUNe. Le premier TUNe (**TUNe0**) déploie les agents *(MA, LA2)* et les instances du serveur *SeD2*. Le deuxième TUNe va administrer l'agent *LA1* et le groupe de serveurs (*SeD1*). **TUNe0** peut déléguer le démarrage des serveurs *SeD1* (*SeD1.start*) à **TUNe1**. Pour cela, il utilise la référence du serveur RMI de **TUNe1** (*Ref_TUNe1*) pour appeler la méthode ***execute*** (figure 6.11).

En résumé, le déroulement du processus d'administration décentralisée est le suivant :

- L'administrateur déploie le premier TUNe (la racine de la hiérarchie) ;
- Chaque TUNe de la hiérarchie déploie ses fils ;

```
public interface TuneIt extends Remote
{
    public void execute (String source, String softwareName, String targetName,
        Object[] args) throws RemoteException ;
    ....
}
```

FIGURE 6.11 – Interface d'un TUNe

- Après le déploiement d'un TUNe, celui-ci commence par le démarrage de son serveur RMI et l'enregistrement de la référence dans RMI registry (qui s'exécute sur son nœud) ;
- Chaque TUNe (ayant au moins un fils) récupère les références des serveurs RMI de ses fils ainsi que les noms des entités logicielles déployées par toute la hiérarchie ;
- Chaque TUNe (sauf la racine) récupère la référence du serveur RMI de son père ;
- La génération de l'architecture Fractal est effectuée ;
- Les TUNe déployés entament le déploiement de l'application proprement dit ;
- Enfin, le démarrage de l'application est effectué par TUNe racine qui se charge de l'exécution du diagramme d'état-transition (startchart) et de la délégation des tâches de démarrage d'une entité logicielle aux TUNe appropriés (chargé de l'administration de l'entité logicielle).

6.3 Personnalisation de la phase d'installation

Après avoir présenté nos contributions sur la décentralisation du processus d'administration, nous montrons comment permettre une personnalisation de la phase d'installation afin de tenir compte de la spécificité de l'environnement d'administration. L'objectif de cette section est de présenter notre approche pour personnaliser le processus d'installation. L'approche est basée sur la description des opérations d'installation et leurs enchaînements. Avec cette description, l'administrateur personnalise le processus d'installation de son application. Nous commençons par présenter notre approche de façon générale. Ensuite nous expliquons comment appliquer cette approche au système d'administration TUNe.

6.3.1 Principe général

Le processus d'installation d'une application est composée d'un ensemble d'activités. Ces activités sont exécutées par le système d'administration et varient généralement en fonction du contexte d'administration. L'installation d'une application consiste à mettre en place un environnement d'administration en créant par exemple des répertoires d'installation sur les nœuds distants ou en effectuant des transferts des paquetages. L'objectif ici est de personnaliser ces différentes activités.

Question : *comment un administrateur peut personnaliser le processus d'installation ?* Cette personnalisation dévient difficile lorsque le processus d'installation est câblé dans le système d'administration.

Pour illustrer la nécessité de personnaliser l'installation d'une application, nous décrivons dans la suite de cette section, le processus d'installation d'une application DIET implanté dans le système d'administration TUNe. Nous montrons ensuite un exemple de déploiement d'une architecture DIET qui nécessite une personnalisation de la phase d'installation. Enfin nous présentons notre approche de personnalisation du processus d'installation.

Processus d'installation d'une architecture DIET

Comme évoqué précédemment, le processus d'installation est *câblé* dans le code source de la plus part des systèmes d'administration notamment dans le système TUNe. Par exemple pour installer des serveurs *SeD*, TUNe exécute les pérations suivantes :

- création du répertoire d'installation sur le nœud alloué à l'entité logicielle ;
- création des démons sur les nœuds. Ces démons sont chargés de l'exécution effective des tâches de déploiement ;
- copie des fichiers binaires et les librairies dans le répertoire d'installation.

Ces opérations généralement codées en *dur* sont automatiquement exécutées par le système d'administration lors du processus de déploiement. L'administrateur n'a donc aucun contrôle sur les opérations d'installation. Par exemple, lorsqu'un serveur NFS est installé, la copie des paquetages sur tous les nœuds n'est plus nécessaire. Il suffit en effet d'effectuer une seule copie des fichiers binaires dans le répertoire NFS.

Pour illustrer nos propos, nous allons décrire un contexte de déploiement d'une application DIET qui nécessite la personnalisation du processus d'installation. Nous disposons d'un répertoire NFS contenant les paquetages (fichiers binaires, libraires) de toutes les entités logicielles (*OMNI*, *MA*, *LA*, *SeD*). Ce répertoire est accessible à partir des nœuds sur lesquels l'application est administrée. Il n'est donc plus nécessaire de créer de répertoire d'installation ou de copier des fichiers binaires sur les nœuds physiques. Des répertoires locaux sont crées sur chaque machine pour la génération des fichiers de configuration et le stockage de fichiers de log. Le processus de déploiement précédent ne nous permet pas d'effectuer l'installation dans ce contexte particulier vu que les opérations d'installation et l'enchaînement de ces opérations sont prédéfinis par le système. Pour remédier à ce problème, nous proposons une personnalisation de la phase d'installation.

Notre proposition est basée sur un langage de description des processus d'installation. L'utilisation d'un langage pour décrire les processus d'installation permet essentiellement aux administrateurs de décrire donc personnaliser l'installation selon leurs besoins. Ce langage fournit une séquence de primitives basiques permettant d'effectuer la création de répertoires, faire de transfert de fichiers ou plus généralement exécuter de commandes pour installer une application. De même il est intéressant de pouvoir définir des primitives pour exécuter des opérations de façon parallèle ou séquentielle (copie séquentielle ou parallèle). Ainsi l'administrateur décrit les processus d'installation de son application. Cette description est interprétée par le système d'administration pour déclencher l'exécution des opérations d'installations sur les machines physiques.

Pour ce type de langage, il est intéressant de prévoir les traitements à effectuer en cas d'erreur lors de l'exécution des opérations d'installation. Cette problématique n'est pas traitée dans cette thèse. Un choix peut être de revenir à l'état initial en défaisant l'opération déjà exécutée [HHW99]. Cela impose d'associer à chaque opération, une autre opération permettant de la défaire. Un autre choix peut être de revenir à un état cohérent prédéfini.

Nous avons vu dans cette section la nécessité de pouvoir personnaliser la phase d'installation et comment donner la possibilité aux administrateurs de personnaliser cette phase du déploiement. Nous avons proposé d'introduire un langage de description des opérations d'installation. Le système d'administration interprète cette description puis exécute les opérations qui la contient.

6.3.2 Application à TUNe

Processus d'installation dans le système TUNe

Le déploiement d'une application dans TUNe commence par l'exécution d'une séquence d'opérations qui se déclenche par le démarrage des composants wrappers Fractal des toutes les instances d'entités logicielles. Les opérations exécutées par ces composants sont :

- allocation d'un nœud au sein de la famille de nœuds *host-family* ;
- création du répertoire d'installation sur le nœud précédemment alloué ;
- copie du *Runtime de TUNe* dans le répertoire d'installation ;
- copie du fichier archive contenu dans l'attribut **software** de la classe UML de l'entité logicielle ;
- décompression de l'archive sur le nœud distant ;
- démarrage du *démon* de TUNe sur le nœud alloué ;
- création d'une liaison RMI entre le composant wrapper de l'entité logicielle et le *démon* de TUNe précédemment créé afin de faire communiquer les composants wrappers de la couche SR et les *démons* de la couche patrimoniale ;
- requêtes *ping* : envoi d'un paquet de données sur le réseau et attente de l'accusé de réception sur le *démon* jusqu'à obtenir une réponse signifiant que la création du s'est bien déroulée.

Une fois que l'exécution de toutes ces opérations sont terminées, TUNe entame le démarrage et la configuration de l'application en exécutant le diagramme de reconfiguration nommé **startchart**.

Notre objectif est d'utiliser un formalisme pour décrire ces opérations afin de permettre d'utiliser des opérations personnalisées, modifier la séquence d'exécution. Pour y parvenir, nous proposons comme formalisme d'installation, les diagrammes état-transition qui sont utilisés par TUNe pour (re)configurer et démarrer une application. Les diagrammes d'état-transition permettent de spécifier une suite d'actions à effectuer, l'ordre de ces actions et le niveau de parallélisation souhaité entre elles. Cela a pour avantage l'utilisation des concepts existants dans TUNe permettant ainsi de minimiser les modifications à effectuer au niveau du code source. Nous introduisons donc un nouveau diagramme dénommé *diagramme d'installation*.

Les opérations du *diagramme d'installation* doivent principalement être effectuées sur les nœuds. Pour pouvoir effectuer des opérations sur les nœuds depuis le diagramme d'état-transition et homogénéiser ainsi l'environnement d'administration, nous proposons d'introduire de wrapper pour les nœuds physiques. Ces wrappers vont permettre à un administrateur d'interagir sur les nœuds au niveau patrimonial depuis le diagramme de reconfiguration du niveau SR. Les nœuds vont donc être administrés comme les entités logicielles. Cela fait l'objet de la section mise en œuvre.

6.3.3 Mise en œuvre dans le système TUNe

Administration des nœuds

L'objectif ici est de pouvoir appeler depuis le *diagramme d'installation*, l'exécution d'une opération d'installation sur un nœud. Nous proposons pour cela, d'introduire un wrapper pour les nœuds. Ce wrapper est décrit dans le formalisme WDL comme les wrappers des entités logicielles et va contenir les opérations d'installation. Les opérations d'installation les plus utilisées sont prédéfinies par défaut notamment (*cp* : copie des ressources logicielles sur une machine, *mkdir* : création d'un répertoire, etc.).

Au niveau SR de TUNe, nous introduisons un attribut supplémentaire **wrapper** pour les clusters définis dans le *diagramme de nœuds*. Cet attribut indique le fichier WDL associé aux nœuds du cluster. Le fichier WDL va permettre l'encapsulation des nœuds du cluster dans des composants Fractal que nous appelons *wrapper node*. Au même titre que les composants wrappers des entités logicielles, ces composants fournissent une interface uniforme définie par le wrapper du fichier WDL du cluster. Cette interface peut fournir diverses opérations : copie de ressources, création ou suppression d'un répertoire ou plus généralement l'exécution d'un script shell nécessaire pour l'installation d'une application, etc.

Au niveau patrimonial, la gestion des nœuds est assurée par un autre démon différent de celui qui gère les entités logicielles (*DEL : démon entité logicielle*). Ce démon que nous appelons *DN : démon node*, est exécuté sur tous les nœuds administrés. Lorsque TUNe alloue un nœud à une instance d'entité logicielle, le *wrapper node* correspondant à ce nœud au niveau SR démarre un démon sur ce dernier au niveau patrimonial et une liaison RMI est créée entre le démon et son composant *wrapper node*. Cette liaison RMI permet la communication entre le composant *wrapper node* qui s'exécute au niveau SR et son démon du niveau patrimonial. Pour exécuter une opération définie dans WDL du cluster d'un nœud depuis le SR, le composant *wrapper node* utilise la liaison RMI pour déléguer l'opération au démon au niveau patrimonial. Le démon va se charger de l'exécution effective de l'opération sur le nœud physique. La figure 6.12 montre l'encapsulation des nœuds et la répartition des démons correspondants.

La figure 6.12 montre une architecture logicielle composée de plusieurs instances d'entités logicielles déployées sur 6 nœuds. Nous remarquons sur la figure que des composants wrappers node sont instanciés au niveau SR afin d'administrer les nœuds au niveau patrimonial à travers les démons. Nous remarquons également que plusieurs instances d'entités logicielles peuvent être déployées sur le même nœud. Cependant un seul *démon node* est crée sur un nœud.

Plusieurs mots clés sont introduits :

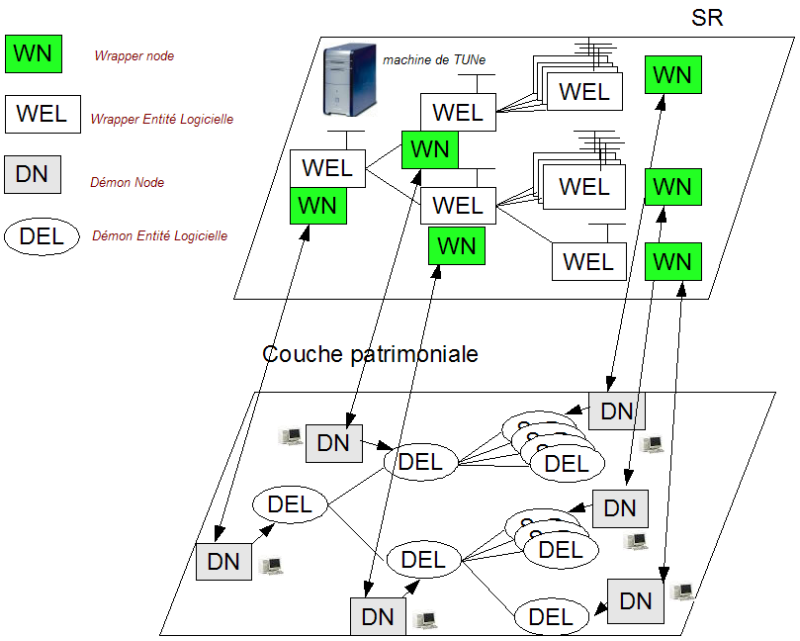

FIGURE 6.12 – Introduction de wrapper pour les nœuds

- Le mot clé ***node*** a été introduit pour accéder au(x) nœud(s) d'une ou plusieurs instances logicielles. Ce mot clé permet d'effectuer une opération d'installation sur un nœud d'une entité logicielle. Par exemple *LA.node.mkdir (« /tmp »)* permet de créer sur les nœuds des toutes les instances *LA* le répertoire */tmp*. *LA.node* est appliqué à une classe *LA* donc retourne un ensemble de nœuds (le nœud de chaque instance de *LA*) :
  - s'il n'existe qu'une seule instance *LA*, *LA.node* retourne un seul nœud ;
  - si plusieurs instances *LA* partagent le même nœud, la redondance est supprimée.
- Pour dire qu'on s'adresse au *diagramme de nœuds* (mot clé ***grid***). L'opération ***grid****.toulouse.mkdir (« /tmp »)* crée le répertoire */tmp* sur tous les nœuds du cluster *toulouse* ;
- Pour dire qu'on s'adresse au *diagramme de configuration* (le mot clé ***config***). Par exemple pour une architecture DIET, l'opération ***config****.MA.LA.SeD.configure* est équivalente à l'opération *SeD.configure* (la configuration des toutes les instances *SeD*). L'opération ***config****.*.configure* permet de configurer toutes les entités logicielles (symbole ***) du *diagramme de configuration*.

Quelques opérations sont définies par défaut dans TUNe notamment :

- ***mkdir*** : cette primitive est appliquée aux nœuds et permet de créer un répertoire. Elle prend en paramètre le chemin absolu du répertoire à créer. L'exécution de *SeDSUM.node.mkDir (« /tmp/dirConfig »)* crée le répertoire */tmp/dirConfig* sur tous les nœuds de toutes les instances de l'entité *SeDSUM*. L'exécution commence tout d'abord par collecter les nœuds des instances de

l'entité *SeDSUM*. Ensuite le répertoire est crée sur chaque instance nœud de la collection ;
- ***cp*** : cette primitive est également appliquée aux nœuds. Elle prend deux paramètres. Le premier paramètre indique le nom de la ressource à copier. Le deuxième paramètre indique le répertoire dans lequel la ressource est copiée. L'exécution de *SeDSUM.node.cp (« sed.tgz », « /tmp/dirBin »)* copie dans le répertoire */tmp/dirBin* les fichiers binaires des serveurs de DIET sur les nœuds de toutes les instances de l'entité *SeDSUM* ;
- ***rmdir*** : cette primitive permet de supprimer un répertoire. Elle a le comportement inverse de la primitive *mkdir* ;
- ***deploy*** : cette primitive applique l'algorithme d'installation défini par défaut par TUNe. Elle s'applique à une entité logicielle. Elle peut prendre un paramètre qui indique le répertoire d'installation. Elle commence par créer soit le répertoire d'installation indiqué dans le paramètre de configuration (*dirLocal*) soit celui passé comme paramètre. Ensuite le paquetage (la valeur du paramètre **software**) de toutes les instances de l'entité d'appelle est copié dans le répertoire crée précédemment. L'exemple *SeDSUM.deploy* crée le répertoire d'installation sur tous les nœuds de toutes les instances de l'entité *SeDSUM*. Le paquetage (*sed.tgz*) est ensuite copié dans le répertoire *dirLocal*.

La figure 6.13 montre un exemple d'un *diagramme d'installation* pour une architecture DIET. Sur cette figure, l'architecture DIET à installer est composée d'une seule instance *MA*, de 50 *LA* et de 500 *SeD* dont 10 *SeD* par *LA*. Toutes les entités logicielles seront installées sur les machines du cluster *toulouse*. Ce cluster a un serveur **NFS** (**Network File System**) accessible aux autres nœuds. Les fichiers de configuration de chaque agent est généré localement. L'installation de l'architecture DIET commence par la création d'un répertoire dans le répertoire NFS pour copier les fichiers binaires de DIET (il faut noter que *MA.node* retourne un seul nœud car il n'existe qu'une seule instance de *MA*). Le processus se poursuit par la création des répertoires locaux pour la génération des fichiers de configuration sur les nœuds de chaque agent. Ensuite, les fichiers binaires compressés (*tgz*) sont copiés et décompressés par l'opération *unzip* définie par l'administrateur dans le fichier WDL du cluster *toulouse* contrairement aux autres opérations qui sont prédéfinies par TUNe (*mkdir, cp*, etc.).

Synthèse : Cette partie contribution porte sur la ***performance*** du système d'administration. Nous avons proposé une approche basée sur la **décentralisation de l'administration** et la **personnalisation de la phase d'installation** du déploiement. La **décentralisation** permet de répartir le coût et la charge du processus d'administration. La personnalisation permet de mettre des stratégies d'installation et de tenir compte de la spécificité de l'environnement d'administration afin d'augmenter la performance (un exemple significatif est la diminution du nombre de copies à effectuer en utilisant l'installation d'un serveur SAMBA ou NFS sur les clusters). Un formalisme est proposé pour décrire l'architecture hiérarchique des systèmes d'administration à déployer ainsi que la répartition des tâches entre ces derniers. Cette description est effectuée dans un diagramme supplémentaire introduit

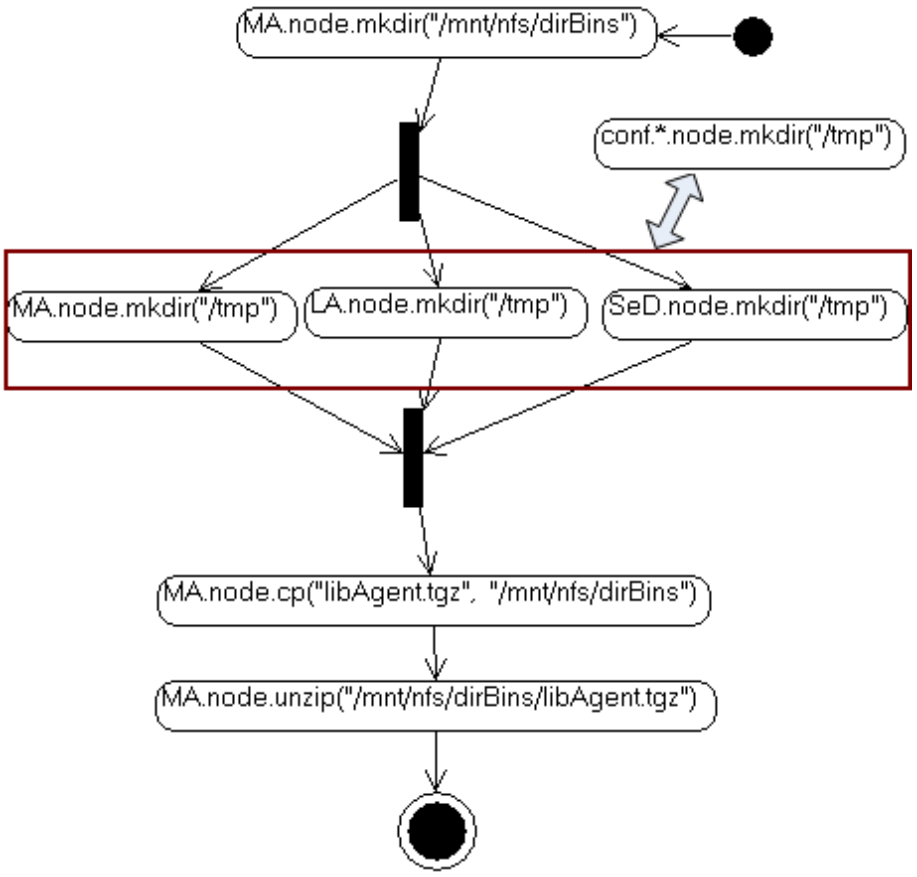

FIGURE 6.13 – Exemple d'un *diagramme d'installation*

dénommé **diagramme d'administration**. Le déploiement des systèmes d'administration est effectué de façon hiérarchique. Un autre diagramme d'état-transition est introduit afin d'exprimer le processus d'installation (**diagramme d'installation**). L'administrateur décrit dans ce diagramme, l'enchaînement des opérations à exécuter pour installer son application. Cela va lui permettre de personnaliser, adapter le processus d'installation selon ses besoins.

Chapitre 7

Hétérogénéité : *Gestion de l'hétérogénéité*

Table des matières

7.1 Rappel du problème

Problème d'hétérogénéité de la grille

On a vu précédemment qu'un des problèmes d'une grille de machines est l'hétérogénéité. Une grille est composée d'un ensemble de clusters distribués sur plusieurs sites indépendants. Les administrateurs de chaque site ont la liberté de choisir les systèmes d'exploitation, les types de matériels en particulier l'architecture processeur des machines, et les politiques de sécurité appliquées aux machines. Les machines d'une grille sont de type divers et sont en grand nombre et physiquement dispersées sur une vaste aire géographique. Elles sont matériellement hétérogènes, en particulier les familles de processeurs et les réseaux d'interconnexions à haute performance. Elles sont également hétérogènes en terme de logiciels, en particulier les systèmes d'exploitation, les bibliothèques de programmation, et les environnements d'exécutions installés. Elles ont des capacités et des performances hétérogènes et variables dans le temps. Cette caractéristique d'hétérogénéité peut engendrer de problèmes lors de l'administration d'une application. Une application peut exister en plusieurs versions qui peuvent être dépendantes du système d'exploitation, de l'architecture processeur, de la quantité mémoire, du disque dur, etc. Le système d'administration

doit installer sur chaque machine, la version de l'application qui lui est compatible selon ses caractéristiques matérielles et logicielles.

Dans la suite de cette partie, nous commençons par faire un bref rappel sur la description de la grille dans TUNe. Ensuite nous présentons une nouvelle description de la grille afin d'exprimer son hétérogénéité (la structure organisationnelle, la composition de la grille en sous cluster, la proximité des clusters, la spécificité de certains nœuds au sein d'un cluster, etc.). Enfin nous présentons notre approche de gestion de l'hétérogénéité lors du déploiement d'une application ainsi que l'application de cette approche et l'implantation au système TUNe.

7.2 L'approche de gestion de l'hétérogénéité

7.2.1 Principe général

TUNe est un système d'administration autonome orienté vers les environnements homogènes de type cluster. Le problème d'hétérogénéités ne se pose pas dans ce type d'environnement vu que toutes les machines ont les mêmes caractéristiques matérielles et logicielles. Cependant lorsqu'on s'oriente vers un environnement de type grille, le problème d'hétérogénéités se pose. En effet un environnement de type grille est composé de plusieurs clusters ayant des caractéristiques différentes, interconnectés par des réseaux différents. Cette caractéristique particulière liée aux grilles engendre de problèmes d'hétérogénéités. Pour remédier à ce problème, nous proposons d'une part une nouvelle description de l'infrastructure matérielle afin d'exprimer sa structure hétérogène. D'autre part nous introduisons la notion de *famille de logiciels*. La description de l'infrastructure matérielle va nous permettre de regrouper les machines selon les caractéristiques communes. Cette description peut être utilisée par le système d'administration lors de l'installation des paquetages.

Une famille de logiciels est un ensemble de logiciels destiné à être installé sur les mêmes types de machines. Elle est caractérisée par un ensemble d'informations nécessaires au bon fonctionnement des logiciels qui la constitue. Parmi ces informations, nous pouvons citer : le système d'exploitation ; le type du processeur (*x86 et x86-64, Itanium, Power, PowerPC, Sparc, Alpha*, etc.). Par exemple, on peut compiler une application DIET pour les machines ayant un système d'exploitation Linux (*Ubuntu*) et un type de processeurs *x86*. Cette compilation constitue une famille de logiciels pouvant être déployée sur un cluster composé que des machines qui s'exécutent sous *Ubuntu* et ayant des processeurs de type *x86*.

Comme évoqué précédemment une grille de calculs est composée d'un ensemble de clusters dispersé sur plusieurs sites. Un cluster est composé d'un ensemble de nœuds homogènes localisés géographiquement dans une même localité (*campus universitaire, centre de calcul, entreprise ou chez un individu*). La description de la

grille va mettre en évidence l'homogénéité des machines. En utilisant cette homogénéité, nous proposons de disposer d'une famille de logiciels par groupe de machines homogènes ou simplement par cluster.

En plus de cette notion de famille de logiciels, nous introduisons la notion de dépôt (**repository**). Un dépôt de logiciels (*software repository*), souvent abrégé en dépôt, est une sorte de base de données où les familles de logiciels sont stockées en vue de leur déploiement par les outils d'administration. Par exemple la plupart des distributions Linux utilisent des dépôts accessibles sur Internet, officiels et non-officiels, permettant aux utilisateurs de télécharger et de mettre à jour des logiciels sous forme de paquets. Un dépôt a un protocole d'accès. Il peut être situé sur une machine locale auquel cas son accès est effectué localement. Il peut également être situé sur une machine distante auquel cas il est accédé par un protocole d'accès spécifique (*ssh, http, svn*, etc.). Nous proposons de décrire les dépôts de logiciels en indiquant les paquetages qu'ils contiennent, les protocoles utilisés pour accéder à ces paquetages. Chaque paquetage est identifié par un *nom symbolique* avec une URL pour y accéder. Ces noms symboliques sont utilisés dans la description des entités logicielles pour identifier le nom de leur paquetage. Nous affectons par ailleurs à chaque cluster (comme paramètre de description), un descripteur des dépôts logiciels lors de la description de l'infrastructure matérielle afin d'indiquer le point d'accès où les paquetages (compatibles avec les nœuds du cluster) peuvent être téléchargés et installés sur ce cluster.

Pour installer le paquetage d'une entité logicielle, le système d'administration :

- identifie le *nom symbolique* noté *NP* du paquetage associé à l'entité logicielle ;
- identifie le cluster noté C sur lequel l'entité logicielle doit être administrée ;
- lit le descripteur du dépôt du cluster C pour retrouver le paquetage *NP* et l'URL correspondante ;
- utilise l'URL précédemment identifiée pour télécharger et installer le paquetage *NP* sur la machine de TUNe (pour la gestion de caches et éviter le téléchargement lors du prochain accès) puis vers la machine cible.

7.2.2 Application au système TUNe

Description de la grille dans TUNe

Comme évoqué dans le chapitre 3 sur le contexte applicatif, TUNe représente la grille comme un ensemble de clusters. Chaque cluster est représenté par une classe UML avec des caractéristiques sous forme d'attributs. Ces caractéristiques sont communes aux nœuds du cluster. Aucune relation de dépendances n'est exprimée entre les clusters.

Question : *Comment peut-on exprimer par exemple que le cluster de* ***toulouse*** *est composé de deux sous clusters ou qu'un nœud d'un cluster est un serveur LDAP donc particulier au sein du cluster ?*

Pour répondre à cette question, il est nécessaire de proposer une nouvelle approche de description de la grille afin de pouvoir exprimer sa topologie et sa structure hétérogène.

Description de la structure hétérogène de la grille

Une grille est par définition composée de plusieurs clusters. Un cluster peut éventuellement être composé des sous clusters dont les nœuds ont certaines caractéristiques communes notamment : le chemin d'installation de java, protocole de connexions à distance utilisable (*ssh*, *oarsh*...) etc. Nous proposons d'introduire la notion de composition dans la description de la grille. Une relation d'association est introduite pour exprimer l'appartenance d'un nœud spécifique au sein d'un cluster. Cette relation est créée entre le nœud spécifique et le cluster auquel il appartient. La notion d'héritage est également introduite. En effet les caractéristiques communes entre les sous clusters qui composent un cluster vont être définies au niveau des caractéristiques du cluster composant. Ainsi parmi les caractéristiques d'un sous cluster, seules celles qui lui sont spécifiques sont à définir. La figure 7.1 montre la description d'une infrastructure matérielle grille composée de trois clusters (**toulouse**, **bordeaux** et **lyon**). Le cluster de **toulouse** est composé de deux sous clusters : **site_irit** et **site_enseeiht**. Deux nœuds du cluster de **bordeaux** ont été nommés (***nodeLog*** et ***nodeRepository***). Les caractéristiques communes aux trois clusters sont définies par la grille. Tous les clusters héritent des caractéristiques de la grille. Cette représentation est simple et garde une cohérence entre la définition d'une grille (composition de clusters) et sa description en profil UML.

Notre approche de description de l'infrastructure matérielle donne une abstraction de la grille mais pas le détail des interconnexions entre les machines, et montre quelles sont les caractéristiques communes aux machines du même sous cluster. Cette information de plus haut niveau et plus synthétique est plus facilement et directement exploitable par le gestionnaire du déploiement.

Gestion de l'hétérogénéité lors du déploiement

Au sein d'un cluster, les machines partagent généralement les mêmes architectures matérielles (processeur) et le même type de système d'exploitation. Ainsi pour gérer l'hétérogénéité, nous avons introduit un nouvel attribut **sourceList** aux caractéristiques prédéfinies d'un cluster. Cet attribut indique (pour un cluster donné) le nom du fichier qui contient l'adresse des dépôts des paquetages compatibles aux nœuds. Chaque ligne du fichier représente l'accès à un paquetage d'une entité logicielle définie dans le *diagramme de configuration* de l'application. Lors de la description d'une entité logicielle, un attribut **software** est défini qui indique le nom de son paquetage (les fichiers binaires et les librairies nécessaires au fonctionnement

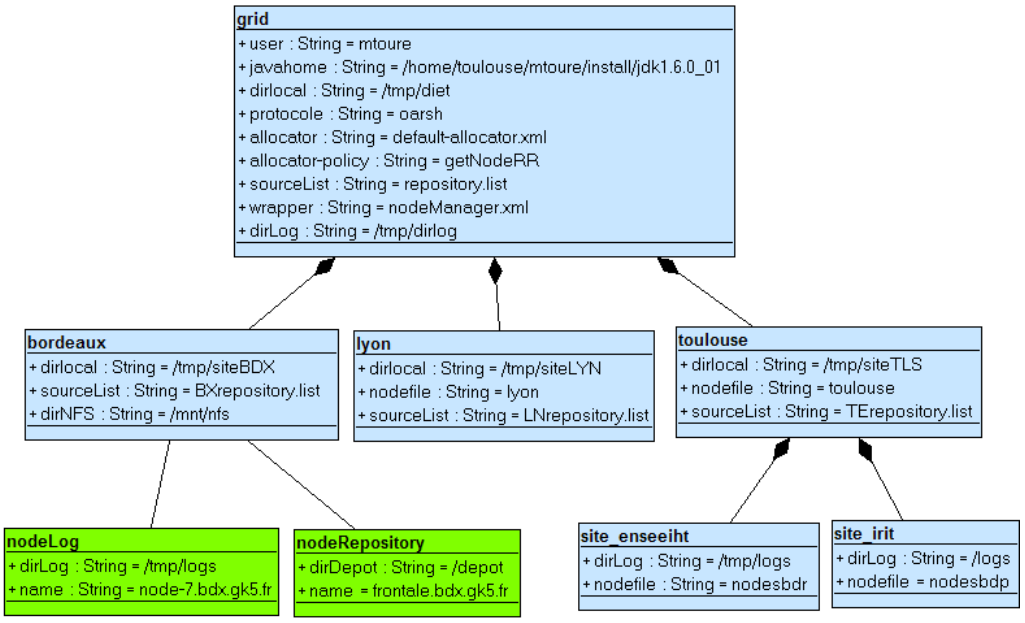

FIGURE 7.1 – Description de la structure hétérogène de la grille

du logiciel). Ce nom est simplement symbolique et associé à une URL dans le fichier **sourceList** du cluster. Cette URL va permettre le téléchargement du paquetage ; du dépôt vers le nœud de l'instance de l'entité logicielle correspondante. La figure 7.2 montre un exemple du contenu d'un fichier **sourceList**. Dans ce fichier, les lignes sont du type :

nom symbolique du paquetage ***protocole :URL***

FIGURE 7.2 – Exemple d'un fichier de dépôt

Pour déployer une instance logicielle :

- un nœud lui est alloué au sein de son **host-family** ;
- la sélection du paquetage compatible au nœud est effectuée à partir de la valeur de l'attribut **software** du logiciel (le *nom symbolique*) et l'adresse de ce dernier dans le fichier indiqué par l'attribut **sourceList** de son **host-family** ;

– le paquetage de l'entité est téléchargé selon le protocole indiqué ou simplement accédé à partir d'un répertoire local (protocole *file*). Le paquetage est ensuite transféré dans un répertoire local à TUNe permettant ainsi de gérer les caches. Avant tout téléchargement d'un paquetage, TUNe consulte d'abord le cache afin de vérifier si le paquetage à télécharger n'est pas présent dans le cache. Si le fichier se trouve dans le cache, un transfert est effectué à partir du répertoire de cache vers le répertoire d'installation du logiciel. Sinon le paquetage est téléchargé selon l'adresse indiquée dans le fichier **sourceList**. L'approche qui consiste à gérer les caches limite le nombre d'accès au dépôt. Ainsi l'accès n'est effectué que si nécessaire. Le schéma de la figure 7.3 illustre nos propos.

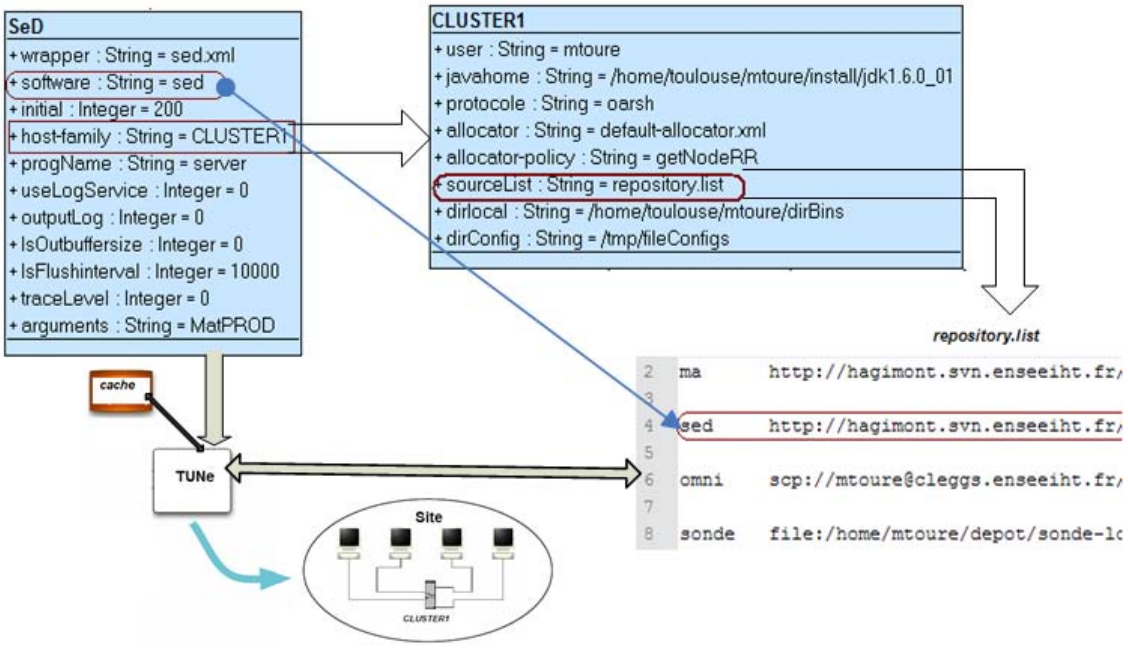

FIGURE 7.3 – Utilisation d'un dépôt

Synthèse : Nous avons présenté une contribution sur la gestion de ***l'hétérogénéités***. Notre approche est basée sur la **description de la structure hétérogène de la grille** et l'**introduction d'un dépôt logiciel (repository)**. Pour appliquer cette contribution à TUNe, nous avons introduit un attribut au niveau de la description des clusters qui va indiquer le fichier de dépôts logiciels. Ce fichier contient l'adresse sous forme URL des paquetages compatibles aux machines du cluster. Nous associons aux entités logicielles, le nom de leur paquetage et la famille de machines sur laquelle elles doivent être installées. Pour installer une application sur les nœuds de la grille, le système d'administration utilise à la fois la description de la grille et le dépôt logiciel des clusters. Les paquetages sont téléchargés selon l'adresse URL et le nom symbolique puis automatiquement installés sur chaque machine administrée.

7.3 Résumé des contributions

Nous avons présenté dans cette partie nos différentes contributions portant sur :

L'expressivité : Nous avons proposé la **description en intension** qui permet de décrire plusieurs instances logicielles avec un minimum de verbosités. Des classes UML sont utilisées pour décrire les classes de logiciels avec la relation d'association pour spécifier les dépendances entre les entités logicielles. Une association peut avoir des cardinalités afin de spécifier la relation de dépendance entre les instances de classe. Un **algorithme de liaison** est proposé pour construire l'architecture logicielle en extension. Cet algorithme utilise la description en intension et la relation de dépendances entre les entités logicielles pour construire la description en extension correspondante.

La performance : Nous proposons pour cette contribution de **décentraliser** l'exécution des opérations d'administration et de pouvoir **personnaliser** le processus d'installation d'une application. La **décentralisation** permet de répartir le coût et la charge du processus d'administration. La **personnalisation** du processus d'installation permet de tenir compte de la spécificité de l'environnement d'administration (la proximité entre les clusters, l'installation d'un serveur SAMBA ou NFS sur les clusters). Cette spécificité peut être utilisée pour par exemple diminuer le nombre de copies et augmenter donc la performance du système d'administration.

L'hétérogénéité : Notre approche de gestion de l'hétérogénéité est basée sur la **description de la grille** afin de spécifier sa structure hétérogène et l'**introduction d'un dépôt logiciel (repository)**. Ce dépôt est utilisé lors de l'installation d'une application pour sélectionner les paquetages compatibles aux machines d'un cluster.

Chapitre 8

Validation

Table des matières

Nous présentons dans ce chapitre les différentes expériences effectuées pour la validation et l'évaluation de nos contributions. Dans un premier temps, nous décrivons le contexte de réalisation de nos expérimentations en présentant le projet grid5000 [CCD+05a]. Les résultats obtenus sont ensuite présentés et analysés. Nous avons réalisé trois types d'expérimentations sur la plate-forme grid5000 afin de valider le passage à l'échelle de notre prototype. Le but de la première expérience est d'administrer une architecture DIET sur un nombre significatif de machines et de valider le passage à l'échelle du formalisme de description que nous avons proposé. La deuxième expérience porte sur la décentralisation de l'administration. L'objectif de cette expérience est de comparer le déploiement effectué par un nombre variable de TUNe afin d'étudier le nombre de systèmes d'administration adéquat pour administrer une application. La dernière expérience porte sur la décentralisation de la reconfiguration. L'objectif est de comparer la reconfiguration *locale* et *distante*. Une *reconfiguration locale* est effectuée localement à un cluster c'est-à-dire l'application à reconfigurer et le système d'administration s'exécutent sur le même cluster. Une *reconfiguration distante* consiste à reconfigurer une application installée sur un cluster (noté C) par un système d'administration qui s'exécute sur une machine d'un autre cluster (différent de C).

Dans la suite de cette section, nous commençons par présenter le projet grid5000 utilisé pour nos expérimentations. Ensuite chaque expérience est présentée avec une analyse des résultats obtenus.

8.1 Expérimentation sur Grid5000

Grid5000 est un projet de grille de calculs dont les clusters sont répartis sur les neuf sites suivants : *Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia* et *Toulouse*. Chaque site comprend un ou plusieurs clusters. Actuellement, Grid5000 dispose d'un total de 1586 nœuds, chacun composé de deux à quatre cœurs (la plupart des nœuds sont bi processeurs, certains processeurs contenant jusqu'à deux cœurs). Au total, il y a donc environ 4384 cœurs. À l'exception d'un des clusters de Grenoble encore en 32 bits, tous les processeurs fonctionnent en 64 bits. On trouve ainsi des processeurs reposant sur l'architecture X86-64 (AMD OPTERON, INTEL XEON), mais aussi sur IBM POWERPC et INTEL ITANIUM2. La connexion réseau entre les clusters est assurée par RENATER. Initialement, les clusters étaient connectés entre eux en 1Gbit/s, mais disposent actuellement d'un débit allant jusqu'à 10Gbit/s. Les machines composant Grid'5000 utilisent GNU/LINUX. La plupart des outils utilisés sont standards. La figure 8.1 montre la répartition des clusters entre les différents sites.

FIGURE 8.1 – Répartition des sites de Grid'5000

Un système NFS est proposé sur chaque site de la grille. Il permet de conserver des données entre deux utilisations de la plate-forme. Le NFS d'un site est accessible à partir de toutes les ressources appartenant à ce site. Cette particularité permet notamment, lors du déploiement d'une application sur un même site, de ne pas transférer explicitement les programmes et les données sur les ressources. Il est en effet possible d'atteindre un fichier en utilisant son adresse unique dans l'arbre du système de fichiers. En revanche, l'espace NFS d'un site est difficilement accessible à partir de ressources localisées sur un site différent.

La plateforme Grid5000 est partagée par plusieurs équipes de recherches qui ont mis en commun un ensemble de ressources constituant la grille. Elle est accessible par plusieurs centaines d'utilisateurs qui peuvent lancer des applications sur la grille (ou une partie) simultanément. La politique d'utilisation actuelle de la plateforme est basée sur la réservation préalable des ressources. En effet un utilisateur de la grille commence tout par réserver des nœuds sur les différents sites qu'il souhaite utiliser. Cela est effectué par un l'outil *OAR* ou *OARGRID*. L'utilisateur programme ses réservations en fonction des nœuds restants disponibles ; le critère d'attribution est « premier arrivé, premier servi »

Toutes nos expériences ont été effectuées sur la plate forme grid5000.

8.2 Expérience à grande échelle

L'objectif ici est d'effectuer une expérience à grande échelle afin de valider notre prototype sur un nombre significatif de machines avec plusieurs entités logicielles à administrer. Une architecture DIET à grande échelle (composée de nombreux agents et serveurs) est administrée sur de centaines de machines de grid5000.

Dans la suite de cette section, nous présentons la réservation des machines avec l'outil *OARGRID* de la plate forme grid5000. Nous décrivons ensuite l'infrastructure matérielle ainsi que la description de l'architecture et les paramètres de configuration de l'application DIET. Cette description de l'application que nous proposons est comparée à celle du système *GoDIET*. Enfin nous effectuons une comparaison entre l'administration décentralisée et centralisée afin de montrer les performances de l'approche décentralisée.

8.2.1 Réservation des machines

L'objectif de cette expérience est de tester notre prototype dans un contexte réaliste à grande échelle, il fallait à cet effet réserver plusieurs centaines de machines. Pour cela, nous avons utilisé l'outil *OARGRID* pour réserver des nœuds sur grid5000. *OARGRID* est un système de réservations open source, développé spécialement pour

la plate forme grid5000. Il permet aux utilisateurs de faire des réservations des machines à l'avance sur plusieurs clusters à la fois. Nous avons pu réserver plus de 700 machines sur différents sites. La figure 8.2 montre la sortie de l'outil *OARGRID* après la réservation.

```
mtoure@fgrelon1:~$ oargridsub bordeaux:rdef="/nodes=150", lyon:rdef="/nodes=50", lille:rdef="/nodes=80", toulouse:rdef="/nodes=50",
nancy:rdef="/nodes=150", orsay:rdef="/nodes=150", sophia:rdef="/nodes=150" -s '2009-11-14 10:00:00' -w '24:00:00'
bordeaux:rdef=/nodes=150, lyon:rdef=/nodes=50, lille:rdef=/nodes=80, toulouse:rdef=/nodes=50, nancy:rdef=/nodes=150, orsay:rdef=/nodes=150,
sophia:rdef=/nodes=150
[OAR_GRIDSUB] [sophia] Date/TZ adjustment: 1 seconds
[OAR_GRIDSUB] [sophia] Reservation success on sophia : batchId = 401418
[OAR_GRIDSUB] [orsay] Date/TZ adjustment: 0 seconds
[OAR_GRIDSUB] [orsay] Reservation success on orsay : batchId = 272465
[OAR_GRIDSUB] [nancy] Date/TZ adjustment: 0 seconds
[OAR_GRIDSUB] [nancy] Reservation success on nancy : batchId = 234850
[OAR_GRIDSUB] [toulouse] Date/TZ adjustment: 0 seconds
[OAR_GRIDSUB] [toulouse] Reservation success on toulouse : batchId = 291191
[OAR_GRIDSUB] [lille] Date/TZ adjustment: 0 seconds
[OAR_GRIDSUB] [lille] Reservation success on lille : batchId = 1004695
[OAR_GRIDSUB] [lyon] Date/TZ adjustment: 0 seconds
[OAR_GRIDSUB] [lyon] Reservation success on lyon : batchId = 280260
[OAR_GRIDSUB] [bordeaux] Date/TZ adjustment: 0 seconds
[OAR_GRIDSUB] [bordeaux] Reservation success on bordeaux : batchId = 819158
[OAR_GRIDSUB] Grid reservation id = 21172
[OAR_GRIDSUB] SSH KEY : /tmp/oargrid//oargrid_ssh_key_mtoure_21172
    You can use this key to connect directly to your OAR nodes with the oar user.
mtoure@fgrelon1:~$ cp /tmp/oargrid//oargrid_ssh_key_mtoure_21172* reservation_orsay/
mtoure@fgrelon1:~$
```

FIGURE 8.2 – Sortie de l'outil *OARGRID* après une réservation

En tout, 7 clusters ont été utilisés dont :

- 150 nœuds sur le cluster de Sophia
- 150 nœuds sur le cluster de Nancy
- 150 nœuds sur le cluster d'Orsay
- 50 nœuds sur le cluster de Lyon
- 80 nœuds sur le cluster de Lille
- 50 nœuds sur le cluster de Toulouse
- 150 nœuds sur le cluster de Bordeaux

La description de cette infrastructure matérielle est présentée sur la figure 8.3. Pour des raisons de lisibilité, les caractéristiques ont été omises. Nous avons regroupé les clusters par site. Les clusters qui sont localisés géographiquement au nord (selon la disposition des clusters sur la carte de France) sont regroupés et forment *siteNord* (*Nancy, Lille, Orsay*) et ceux qui sont localisés au sud forment *siteSud* (*Bordeaux, Toulouse, Sophia*). Le cluster de *Lyon*, qui est relativement au centre, est isolé.

Nous constatons que notre formalisme permet parfaitement de décrire la topologie et la structure de l'infrastructure matérielle. Les clusters sont regroupés par proximité géographique ; ce qui n'était pas possible avec la description de la grille proposée par TUNe.

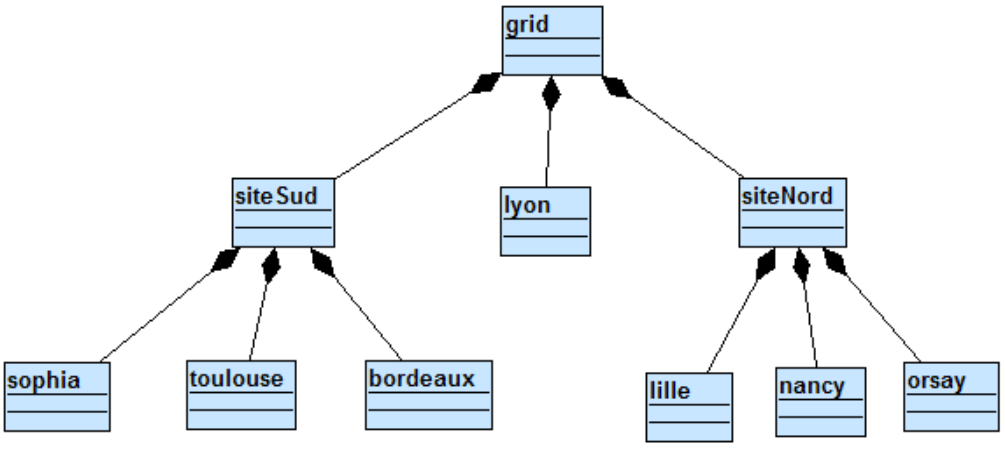

FIGURE 8.3 – Description de la grille d'expérimentation

8.2.2 Architecture de l'application administrée

Pour expérimenter, nous avons utilisé une application DIET composée de 800 *SeD*, 40 *LA*, 1 *MA* et 1 *omniNames*. L'entité *MA* est reliée aux 40 agents locaux (*LA*). Chaque *LA* est relié à 20 *SeD*. Pour déployer cette application, il faut tout d'abord décrire l'architecture logicielle et les paramètres de configuration de chaque entité logicielle. Ainsi grâce à la description en intension et l'algorithme de liaisons, nous avons utilisé que 5 classes UML pour effectuer la description de l'application. La figure 8.4 montre cette description. Pour des raisons de lisibilité, tous les paramètres de configuration ne sont pas mentionnés.

Pour montrer l'efficacité et la simplicité de notre approche, nous avons comparé notre description à celle utilisée par *GoDIET*. La figure 8.5 montre le contenu du fichier de description de GoDIET composé de plus de **10.000 lignes** pour décrire la même application.

8.2.3 Administration

Pour administrer l'architecture DIET précédemment décrite, 3 TUNe ont été utilisés. Le premier TUNe se charge de l'administration des agents (*MA* et *LA*) et *d'omniNames*. Les deux autres TUNe sont chargés d'administrer les serveurs avec chacun un site (*siteNord* et *siteSud* dont 400 *SeD* par site). La figure 8.6 montre la hiérarchie des TUNe déplyés.

Pour mesurer la performance de notre prototype en terme du temps de déploiement, nous avons mesuré le temps mis par trois TUNe et le temps mis par un seul TUNe pour déployer l'architecture DIET. Pour les 3 TUNe, le temps de déploiement

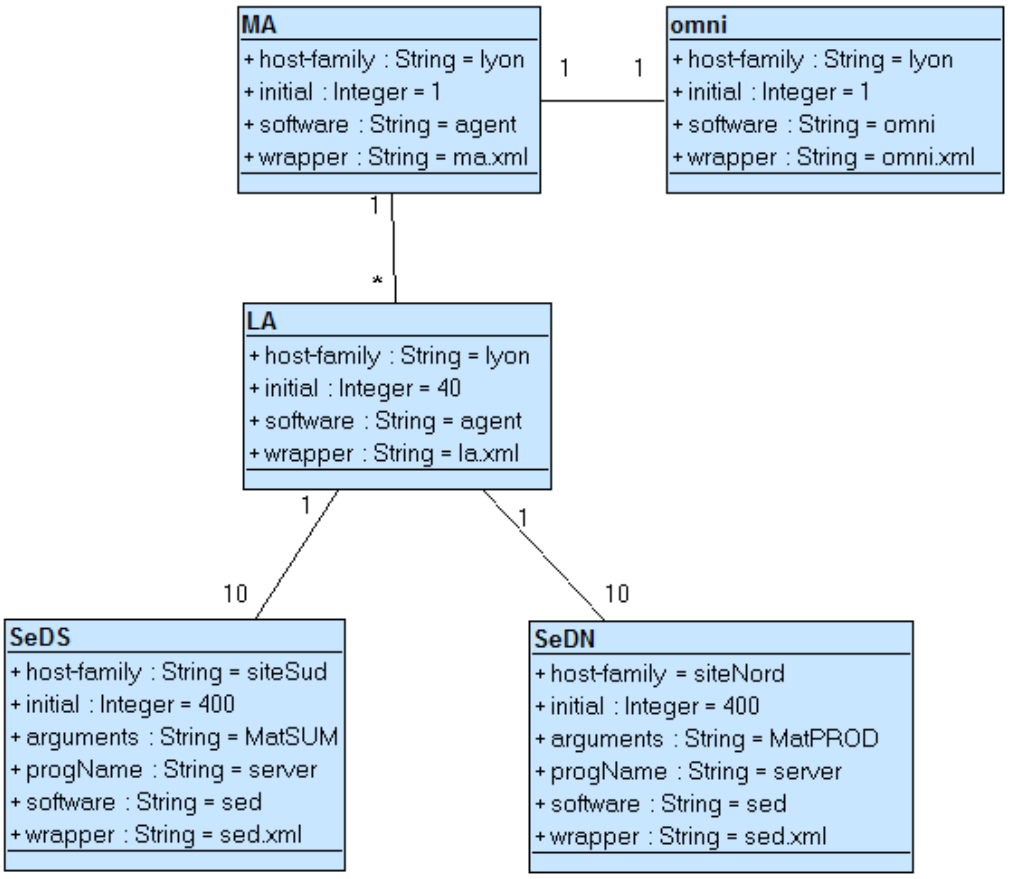

FIGURE 8.4 – Description de l'infrastructure logicielle

est estimé à **110 secondes** (moins de 2 minutes). Tandis qu'avec un seul TUNe, le temps du déploiement est estimé à plus de **350 secondes** (plus de 5 minutes).

Synthèse : L'expérience à grande échelle menée dans cette section nous a montré la facilité et la simplicité au niveau de l'expressivité ainsi que la performance de notre approche d'*administration décentralisée*. Nous avons pu décrire une architecture DIET composée de plus de 840 entités logicielles. Contrairement à des outils comme *GoDIET* qui demande des milliers de lignes XML pour décrire cette architecture DIET, notre prototype ne demande que 5 classes UML pour décrire l'architecture logicielle. Nous avons également montré la performance de TUNe hiérarchisé pour un déploiement à grande échelle. Avec trois TUNe, nous constatons un gain de plus de 50% sur le temps de déploiement par rapport au déploiement d'un seul TUNe sur plus de 700 machines.

```
<diet_services>
<!-- Description d'omniNames-->
</diet_services>
<diet_hierarchy>
<!-- Les agents et les serveurs Diet. -->
        <master_agent label="MA" useDietStats="1">
             <config server="host1"
                 trace_level="1"
                 useLogService="1"
                 lsOutbuffersize="10000"
                 remote_binary="dietAgent"/>
                 <local_agent label="LA_0" useDietStats="0">
                    <config server="host2"
                          trace_level="1"
                          useLogService="1"
                          lsOutbuffersize="10000"
                          remote_binary="dietAgent"/>
                   <SeD label="SeDS_0">
                          <config server="siteSud_host2"
                                  remote_binary="server"/>
                                  <parameters string="MatSUM"/>
                   </SeD>
                 .
                 .
                 .
                 .
                 <local_agent label="LA_39" useDietStats="0">
                   <config server="lyon_host50"
                          trace_level="1"
                          useLogService="1"
                          lsOutbuffersize="10000"
                          remote_binary="dietAgent"/>
                 ...

                    <SeD label="SeDN_399">
                         <config server="siteNord_host340"
                         remote_binary="server"/>
                         <parameters string="MatPROD"/>
                    </SeD>
        </local_agent>
    </master_agent>
  </diet_hierarchy>
```

FIGURE 8.5 – Description de l'application par GDIET

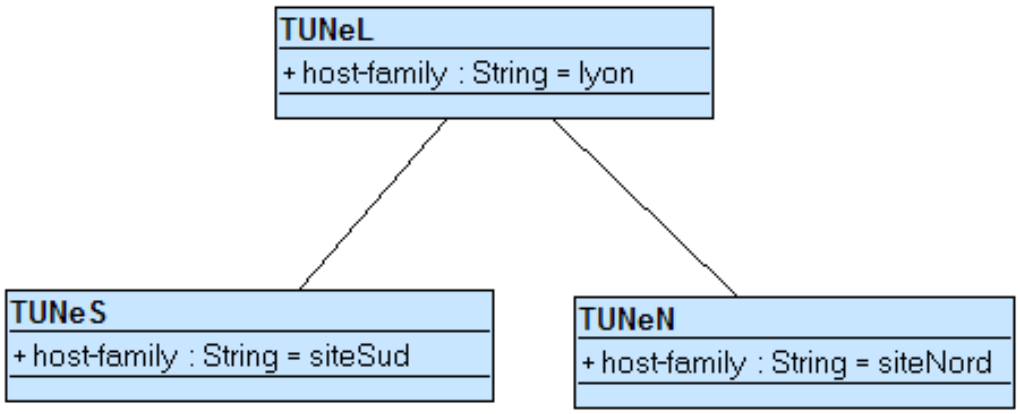

FIGURE 8.6 – Hiérarchie des TUNe utilisée pour l'expérimentation

8.3 Déploiement décentralisé et hiérarchique : variation du nombre de TUNe

Nous avons précédemment mené une première expérience afin de montrer l'efficacité de la hiérarchisation de TUNe. Nous étudions dans cette section dans une première expérience, le nombre de TUNe nécessaires pour administrer une application. Grâce au *diagramme d'installation* introduit pour décrire le processus d'installation, une deuxième expérience a été menée afin de prendre en compte la spécificité de la grille (installation de serveurs NFS sur les clusters.). Nous avons vu que la phase d'installation demande beaucoup de ressources à la machine qui administre, l'objectif de cette deuxième expérience est de connaître l'influence des copies sur le temps de déploiement. Le contexte de l'expérience est composé de plus de 320 nœuds dont :

- 110 sur le cluster de Nancy (cluster Grelon),
- 50 sur le cluster de Sophia (cluster Azur),
- 40 sur le cluster de Lyon (cluster Sagitaire),
- 48 sur le cluster de Bordeaux (cluster Bordeplage),
- 79 sur le cluster de Bordeaux (cluster Bordereau).

La description de l'infrastructure matérielle est représentée par la figure 8.7

Une application DIET composée d'1 *omniNames*, d'1 *MA*, 4 *LA* et 320 *SeD* dont chaque *LA* est relié à 80 *SeD*. La figure 8.8 montre les résultats obtenus sans utilisation de NFS. La figure 8.10 montre le résultat de l'expérience obtenu en tenant compte de la spécificité des clusters (installation de serveur NSF sur les clusters).

Nous nous sommes intéressés à l'évaluation de la performance du déploiement hiérarchique. Le résultat de l'expérience montre le gain de notre approche en terme du temps de déploiement. Le passage d'un TUNe à deux, divise le temps de déploie-

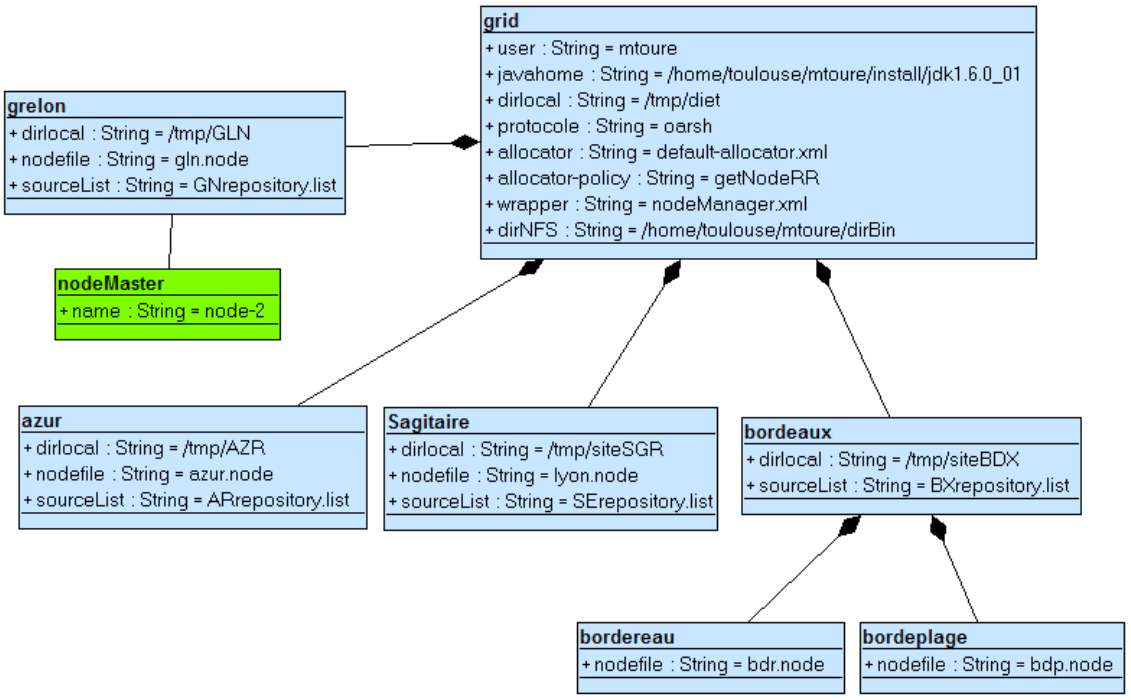

FIGURE 8.7 – Description de la grille

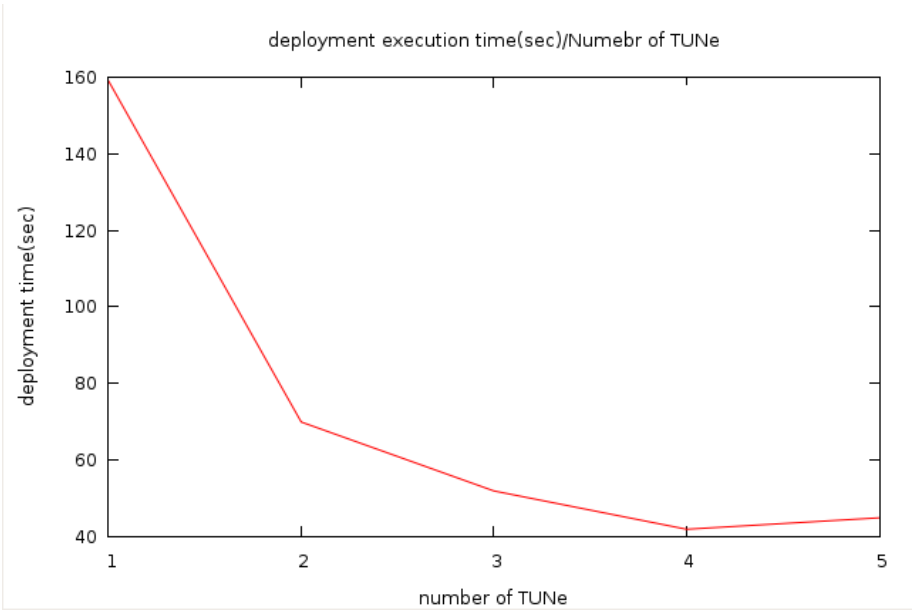

FIGURE 8.8 – Temps du déploiement en fonction du nombre TUNe déployés

ment par deux (plus de 50% de gain). A partir d'une hiérarchie de 5 TUNe, nous observons que le temps de déploiement a légèrement augmenté. Cette augmentation est due au nombre de TUNe disproportionné par rapport au nombre de nœuds et le nombre d'instances logicielles à administrer. Il est donc plus commode de limiter

le nombre de TUNe selon le nombre de nœuds ou le nombre d'instances d'entités logicielles à déployer.

Nous avons également effectué une expérience en tenant compte de l'installation d'un serveur NFS sur les clusters. Dans ce contexte, le nombre de copies des fichiers binaires est limité au nombre de serveurs NFS. Nous avons donc tenu compte de la spécificité de l'environnement d'administration. La figure 8.9 suivante montre le *diagramme d'installation* utilisé pour cette expérience.

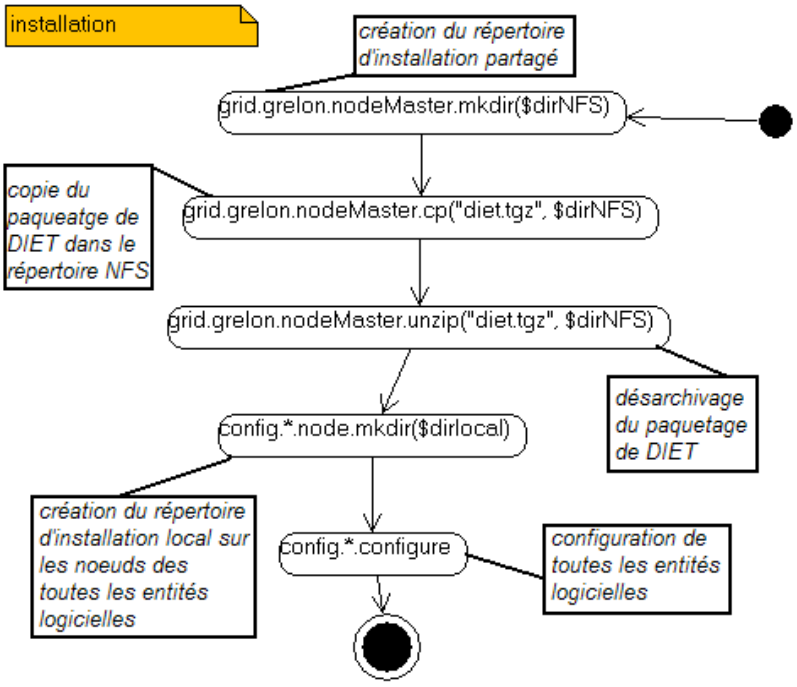

FIGURE 8.9 – Diagramme d'installation

La spécificité de l'environnement est la suivante : tous les clusters ont un serveur NFS, et certains sites notamment *sagitaire*, *azur*, *bordeaux* (*bordeplage et bordereau*) ont au préalable le répertoire d'installation et les fichiers (binaires et libraires) de DIET installés sur leurs machines. Excepté le cluster *grelon*, le répertoire d'installation existe sur chaque cluster ainsi qu'une copie des binaires et des librairies de l'application DIET. Il est donc inutile d'effectuer la création du répertoire d'installation des binaires et la copie des binaires pour les machines de ces clusters. Pour le cluster *grelon*, le *diagramme d'installation* de la figure 8.9 commence par créer le répertoire d'installation dans le répertoire NFS en utilisant le nœud spécifique *nodeMaster* (ce nœud représente un point d'accès au cluster). Ensuite une copie du paquetage DIET (*diet.tgz*) est effectuée dans le répertoire crée précédemment. Il ne reste plus qu'à décompresser ce fichier *tgz* avec la primitive *unzip*. Les fichiers de configuration seront générés dans les répertoires locaux afin de ne pas surcharger le serveur NFS. La suite de l'opération d'installation va créer le répertoire local (*dirlocal*) sur chaque nœud des instances de toutes les entités logicielles (d'où l'utilisation du mot clé ***config***). Enfin toutes les instances logicielles sont configurées.

Cette dernière opération a pour effet de générer les fichiers de configuration dans les répertoires locaux. La figure 8.10 montre les résultats obtenus. Nous observons un gain de plus 60% sur le temps de déploiement le passage d'un TUNe à une hiérarchie composée de 4 TUNe. Ce gain est dû à la fois à la décentralisation de l'administration et à la prise en compte de l'installation d'un serveur NFS qui a diminué le nombre de copies à effectuer.

Nombre de TUNe	Temps d'exécution du déploiement (seconde)
1 TUNe	63
4 TUNe	25

FIGURE 8.10 – *Déploiement avec NFS*

8.4 Reconfiguration

Pour les expérimentations sur la reconfiguration, nous avons déployé une architecture DIET composée d'un *LogService* (*logCentral* et *logTool*), un *MA*, un *LA* et un ensemble de serveurs *SeD*. Une sonde pour l'agent *LA* utilise le *logService* pour détecter les pannes et estimer le temps de réparations. Deux types de *SeD* ont été utilisés : le premier type fait de sommes matricielles (*SeDS*) et le second effectue de produits matriciels (*SeDM*). Nous avons réservé une centaine de machines (précisément 115) sur trois clusters différents :

- 30 machines sur le cluster de Lille
- 50 machines sur le cluster de Nancy
- 35 sur le cluster de Bordeaux.

Les agents *LA* et *MA* sont déployés sur le cluster de *Bordeaux*. Chaque type de serveurs *SeD* est déployé sur un cluster. Ainsi toutes les instances *SeDS* sont déployées sur le cluster *Nancy* et les serveurs *SeDM* sont déployés sur le cluster *Lille*. La figure 8.11 montre l'architecture de l'application utilisée. Nous avons omis les paramètres de configuration pour des raisons de lisibilités.

Pour une architecture DIET, lorsqu'une panne survient (par exemple la panne d'un *LA*), TUNe est chargé de réparer cette panne. Pour expliquer comment par exemple une panne d'un *LA* doit être réparée, nous rappelons le processus de démarrage d'une architecture DIET.

Une architecture DIET est démarrée en suivant une hiérarchie, chaque agent est connecté à son père. Chaque agent s'enregistre dans l'annuaire d'*omniNames* afin qu'il soit retrouvé par les autres agents (à travers son nom). Le *MA* est le premier agent démarré après le démarrage du service de nommages. Il est alors en attente de la connexion d'un agent ou de requêtes en provenance des clients. Ensuite les *LA* sont démarrés et vont s'inscrire auprès du *MA*. Deux types d'agents peuvent alors

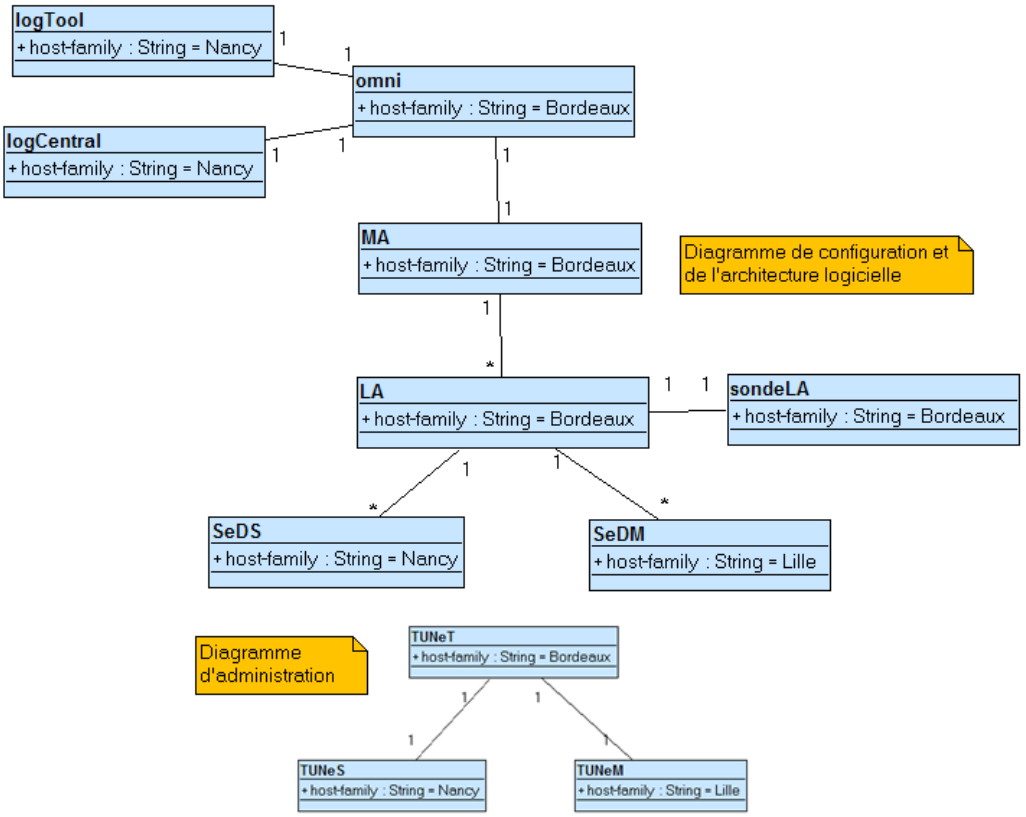

FIGURE 8.11 – Les diagrammes (*reconfiguration et d'administration*) utilisés pour expérimenter la reconfiguration

être connectés aux *LA* : soit des *SeD* soit d'autres *LA* pour ajouter un niveau de hiérarchie à la branche.

Avec cette architecture hiérarchique, lorsqu'un *LA* tombe en panne, il faut le redémarrer et redémarrer toute la hiérarchie de serveurs *SeD* en dessous (du *LA*). Cela est nécessaire afin que chaque *SeD* puisse se réenregistrer au près de leur *LA*.

Afin de pouvoir tester une reconfiguration locale et distante, nous avons effectué deux expériences. La première expérience, nous avons déployé 1 TUNe sur le cluster de *Bordeaux* qui est chargé d'administrer toute l'architecture DIET avec une sonde pour surveiller l'agent *LA*. Si l'agent *LA* tombe en panne, la sonde va détecter cette panne et lance une notification à TUNe afin qu'il exécute un diagramme de reconfiguration de réparation. La figure 8.12 montre ce diagramme de réparation nommé *repairLA*. Pour simuler une panne, nous avons *tué* le processus du *LA*.

Ce diagramme commence par arrêter la sonde (***this.stop***) afin qu'elle génère plus d'autres notifications. L'agent ***LA*** va être démarré (***LA.start***). Les serveurs ***SeD*** liés au ***LA*** vont tous être redémarrés. Cela consiste à stopper tous les serveurs (***SeDS.stop, SeDM.stop***) puis les démarrer (***SeDS.start, SeDM.start***). Ce type de reconfiguration est appelé la *reconfiguration distante*. En effet TUNe installé

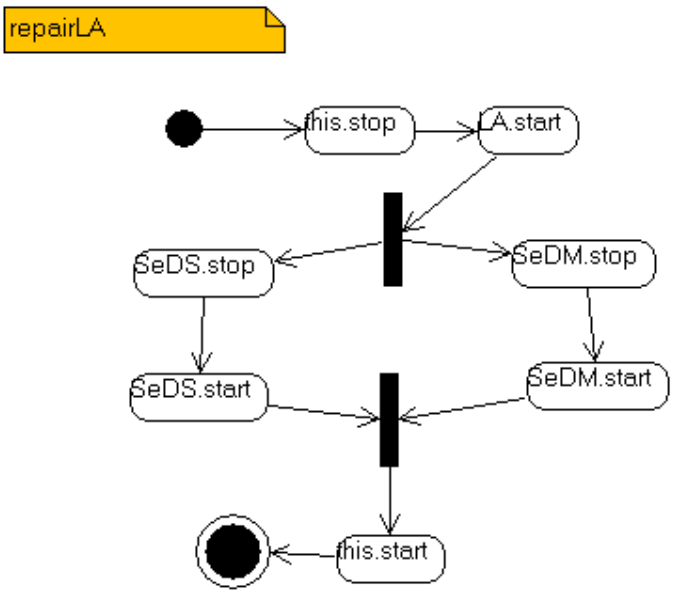

FIGURE 8.12 – Diagramme de réparation d'un LA

sur une machine du cluster **Bordeaux** effectue un appel distant de redémarrage des serveurs situés sur les deux autres clusters (**Nancy et Lille**) (voir figure 8.13).

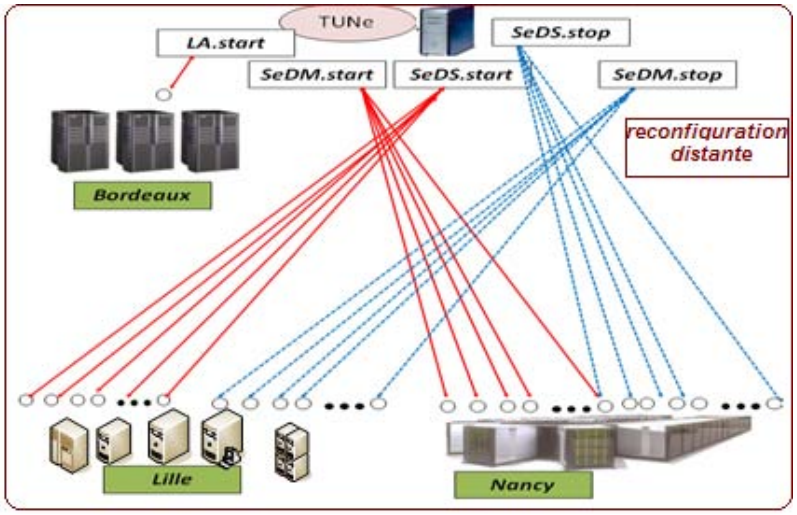

FIGURE 8.13 – Reconfiguration distante

Pour la deuxième expérience, nous avons déployé une hiérarchie de 3 TUNe. Chaque TUNe est déployé sur un cluster différent (*Bordeaux, Lille* et *Nancy*). Dans ce cas si *LA* tombe en panne, la sonde va détecter la panne et va s'adresser au TUNe qui l'administre (TUNeT) afin d'exécuter le diagramme de reconfiguration pour réparer la panne. TUNeT commence l'exécution du diagramme par l'arrêt de la sonde (*this.stop*). La sonde étant administrée par TUNeT, l'action *this.stop* est exécuté par ce dernier. L'action suivante (*LA.stop*) est également exécutée par

TUNeT. Cependant TUNeT transmet par un appel RMI les deux autres actions *SeDS.stop* et *SeDM.stop* aux TUNeS et TUNeM afin que chacun stoppe ses serveurs. Pour le démarrage, un autre appel RMI est effectué par TUNeT pour demander aux autres TUNe (TUNeS et TUNeM) de démarrer leurs serveurs.. Cela entraîne une *reconfiguration locale*. Les appels de redémarrage (*stop* et *start*) des serveurs sont effectués localement. La figure 8.14 montre la *reconfiguration locale*.

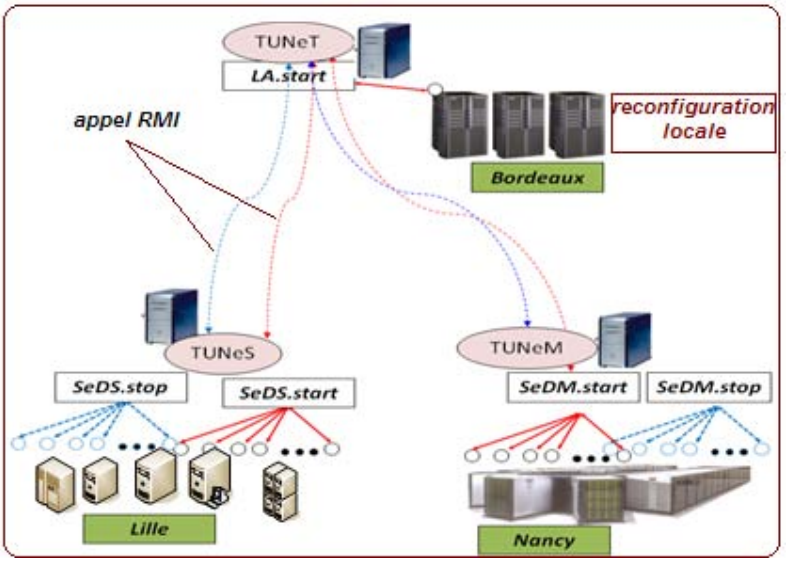

FIGURE 8.14 – Reconfiguration locale

Pour la reconfiguration locale, 2 appels RMI sont effectués par TUNeT au TUNeS afin qu'il redémarre les serveurs SeDS de façon locale sur le cluster de *Lille*. Le premier appel déclenche l'arrêt des *SeDS* par TUNeS. Le second appel effectue le démarrage des serveurs. Le même processus est exécuté par TUNeM sur le cluster de *Nancy*. Ce dernier reçoit les deux appels RMI de TUNeT et se charge de l'exécution des opérations *stop* et *start* sur les instances *SeDM*. L'exécution est effectuée de façon locale par TUNeM sur le cluster *Nancy*.

Les latences de ces deux types de reconfigurations dépendent du temps d'aller-retour d'un paquet entre deux machines. Dans [SLAS07], des mesures (figure 8.15) répétées ont montré une latence de connexions intra cluster de la grille grid5000 (i.e., entre les nœuds d'un même cluster) de l'ordre de 0,05 ms, ainsi qu'une la latence de connexions inter cluster variant de 6 à 20 ms suivant les villes (voir figure 8.15). Ces mesures pratiquement constantes font ressortir un rapport de l'ordre de 10^2 entre les latences inter-cluster et intra-cluster.

Les résultats obtenus pour les reconfigurations sont représentés sur la figure 8.16.

Le temps de reconfiguration dépend du nombre de *SeD* reliés au *LA*. Nous constatons que pour TUNe centralisé (un seul TUNe qui administre l'application), le temps de reconfiguration devient explosif lorsque le nombre de *SeD* commence à dépasser

from \ to	orsay	grenoble	lyon	rennes	lille	nancy	toulouse	sophia	bordeaux
orsay	0.034	15.039	9.128	8.881	4.489	95.282	15.556	20.239	7.900
grenoble	14.976	0.066	3.293	15.269	12.954	13.246	10.582	9.904	16.288
lyon	9.136	3.309	0.026	12.672	10.377	10.634	7.956	7.289	10.078
rennes	8.913	15.258	12.617	0.059	11.269	11.654	19.911	19.224	8.114
lille	10.000	10.001	10.001	10.001	0.001	10.001	20.000	20.001	10.001
nancy	5.657	13.279	10.623	11.679	9.228	0.032	98.398	17.215	12.827
toulouse	15.547	10.586	7.934	19.888	19.102	17.886	0.043	14.540	3.131
sophia	20.332	9.889	7.254	19.215	16.811	17.238	14.529	0.051	10.629
bordeaux	7.925	16.338	10.043	8.129	10.845	12.795	3.150	10.640	0.045

FIGURE 8.15 – Latences moyenne entre les clusters de Grid5000[SLAS07]

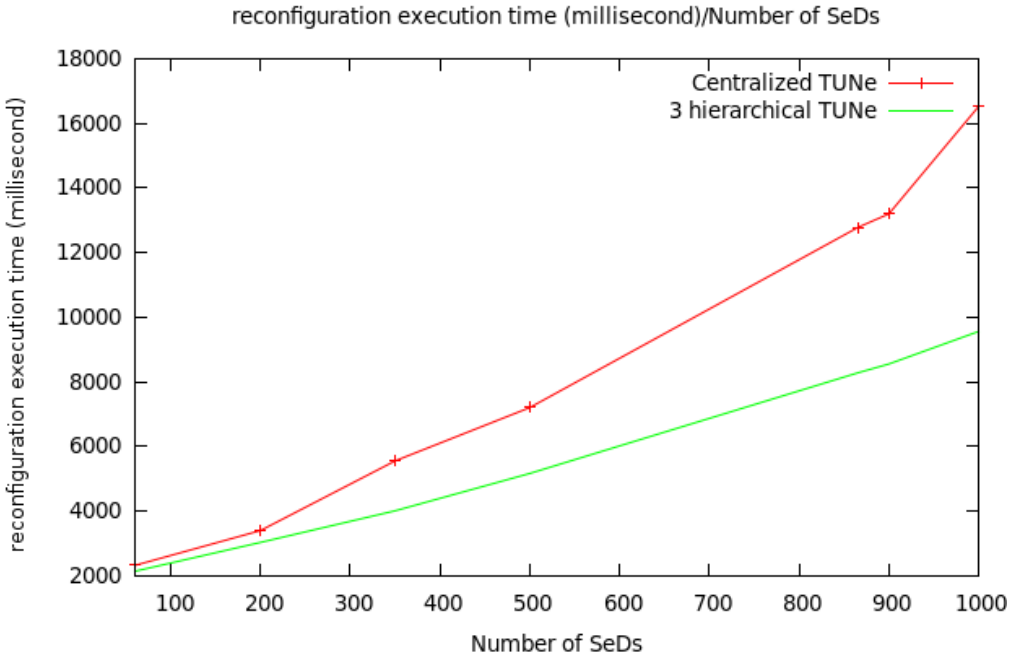

FIGURE 8.16 – Mesure du temps de reconfiguration en fonction du nombre de serveurs (SeD)

1000 instances. Pour TUNe hiérarchisé, le temps de reconfiguration augmente lentement. Lorsque le nombre de serveurs atteint 1000, nous constatons un gain de 50% sur le temps de reconfiguration par rapport au TUNe centralisé.

Synthèse : Nous avons présenté dans ce chapitre les différentes expérimentations pour valider notre prototype. Trois types d'expérimentations sont effectuées :

– Une architecture DIET composée de plus 800 serveurs *SeD*, 40 *LA* et d'1 *MA* a été déployée sur 700 machines réparties sur 7 sites de l'infrastructure grid5000. L'objectif de cette expérience est d'utiliser un nombre significatif de machines et d'instances logicielles. Nous avons montré que la description de l'architecture DIET déployée a été effectuée avec un minimum de verbosités

contrairement au système *GoDIET* qui nécessite un fichier de description de plus **10.000 lignes**. Avec l'utilisation de 3 TUNe pour déployer l'application, le temps de déploiement est estimé à **110 secondes**. Tandis qu'avec un seul TUNe, le temps de déploiement est estimé à **350 secondes** (plus de 50% de gains) ;

- L'objectif du deuxième type d'expérimentation est d'étudier le nombre de TUNe adéquats pour administrer une application selon le nombre d'instances d'entités logicielles et le nombre de machines à administrer. Nous avons déployé une application DIET en faisant varier le nombre de TUNe. Cette variation nous a permis de conclure que la performance de l'administration décentralisée dépend à la fois du nombre de machines utilisées et le nombre d'instances d'entités logicielles de l'application administrée ;
- Le dernier type d'expérimentation porte sur la reconfiguration. Nous avons comparé deux types de reconfigurations : *la reconfiguration locale* et *la reconfiguration distante*. 1 TUNe a été utilisé sur le cluster de *Bordeaux* pour administrer une architecture DIET sur les machines de 3 clusters : *Bordeaux, Lille, Nancy*. La panne d'un *LA* nécessite le redémarrage de plusieurs *SeD* déployés sur deux autres clusters *Lille* et *Nancy*. Nous avons constaté que le temps mis par TUNe pour réparer la panne s'explose lorsque le nombre de *SeD* dépasse 1000. Lorsque 3 TUNe sont déployés avec un TUNe par cluster (*Bordeaux, Lille, Nancy*), le temps de réparation devient stable dû à la reconfiguration locale (redémarrage local de *SeD*) effectuée par les TUNe déployés sur les clusters *Lille, Nancy*.

Chapitre 9

Conclusion et perspectives

Table des matières

9.1 Conclusion

Le calcul intensif est un champ d'application de l'informatique qui croît sans cesse et qui est largement utilisé dans de nombreux domaines scientifiques. Afin de simuler des phénomènes toujours plus complexes (par exemple la météo) et de disposer de résultats d'une précision croissante, d'importants moyens de calculs doivent être utilisés. Parmi les moyens de calculs utilisés, on peut citer les grilles. Une grille est composée d'un ensemble de clusters. Un cluster regroupe de machines homogènes situées dans le même endroit géographique. Les grilles de calcul sont une cible de choix pour les applications de simulation numérique compte-tenu de la puissance de calcul qu'elles offrent. Cependant, il est nécessaire d'avoir des outils appropriés qui se chargent de l'administration (déploiement, observation, adaptation à l'exécution) de telles applications. Toutefois les ressources d'une grille sont plus complexes à utiliser que les stations de travail ou les clusters de machines. Cela est dû à la dynamicité des nœuds, à leurs hétérogénéités et à l'échelle qui empêche les administrateurs d'avoir une connaissance globale de l'environnement d'administration. De nombreux projets ont été menés dans le contexte d'administration à grande échelle. L'étude menée dans l'état de l'art de cette thèse a montré que les propositions sont incomplètes et souvent spécifiques à des types d'applications. Nous avons proposé des solutions génériques, qui tiennent compte du contexte d'échelle. Le travail de cette thèse s'inscrit dans le cadre du projet TUNe,un système d'administration autonome d'applications réparties. Notre objectif est d'étendre TUNe pour le faire passer à l'échelle. Les solutions

proposées sont donc évaluées avec le système TUNe. Nous avons proposé quatre contributions organisées dans trois chapitres sur les problèmes : *d'expressivités*, *de performances* et *d'hétérogénéités*.

La première contribution porte sur *l'expressivités*. Une approche basée sur la **description en intension** est proposée permettant de tenir compte du contexte d'échelle. Cela nous a permis de décrire la configuration de centaines d'entités logicielles avec un minimum de verbosités. Nous avons également proposé un algorithme dénommé **algorithme de liaison** pour instancier des patterns d'architectures logicielles. L'administrateur décrit l'architecture de l'application en intension avec des contraintes sur les cardinalités. L'algorithme génère l'architecture en extension. Cette architecture est ensuite utilisée par TUNe pour construire le SR composé que des composants Fractal.

Nous avons proposé la **décentralisation de l'administration** et la **personnalisation du processus d'installation** comme deuxième contribution pour remédier au problème de *performances* lors de l'administration d'une application. La **décentralisation de l'administration** répartit la charge et le coût que peut engendrer le processus d'administration. Avec la personnalisation de la phase d'installations, l'administrateur peut mettre des stratégies en place pour améliorer la performance car c'est la phase la plus couteuse (connexion, copie, etc.) lors du déploiement d'une application. Nous avons présenté un **déploiement hiérarchique** de TUNe. La hiérarchie de TUNe est décrite sous forme arborescente (**diagramme d'administration**). Chaque nœud de cette arborescence représente un TUNe. Chaque TUNe va se charger d'une part du déploiement de ses fils (des TUNe), d'autre part de l'administration d'une partie de l'application. Nous avons introduit la notion de wrapper pour les nœuds afin de pouvoir les administrer. Le processus d'installation est décrit dans un **diagramme d'installation**. Ce diagramme contient la séquence d'opérations à exécuter pour installer l'application.

Enfin la dernière contribution porte sur la *gestion d'hétérogénéités* de l'infrastructure matérielle et logicielle lors du déploiement. Nous avons proposé une **description de la topologie de l'infrastructure matérielle,** qui permet de décrire des environnements complexes. Ce modèle de description permet de représenter la structure hétérogène d'une grille. Une grille est représentée par une composition de clusters. Un cluster peut être composé par des sous clusters. Nous avons également proposé de décrire des nœuds spécifiques d'un cluster. Cela permet aux administrateurs de désigner et nommer des nœuds au sein d'un cluster afin de les utiliser par exemple dans les diagrammes de reconfiguration. Pour gérer l'hétérogénéité lors de l'installation d'une application, nous avons introduit la notion de dépôt qui va contenir les différentes versions des entités logicielles. Un nouvel attribut (**sourceList**) est ajouté aux caractéristiques d'un cluster dont la valeur indique le nom du fichier contenant l'adresse des dépôts des entités logicielles compatibles avec les nœuds du cluster. Lors du déploiement d'une instance logicielle (notée ***IEL***), le fichier **sourceList** du **host-family** de ***IEL*** est utilié afin de sélectionner son paquetage et l'installer sur son nœud de déploiement.

Nous avons mené des expériences pour valider nos contributions. Trois types d'expérimentations ont été menées pour valider le passage à l'échelle de notre prototype. La première expérience a pour objectif d'effectuer l'administration d'une application de grande taille (composée de nombreuses entités logicielles) sur un nombre important de machines. Ainsi nous avons réservé plus de 700 nœuds sur l'infrastructure Grid5000. Une application DIET composée de plus de 800 instances d'entités logicielles a été décrite et administrée. Cette expérience nous a permis de conclure que notre prototype passe à l'échelle au niveau d'expressivités et de performances. Nous avons effectué une deuxième expérience afin d'étudier le nombre adéquat de TUNe à déployer pour administrer une application et de valider la personnalisation du processus d'installation. Grâce à la personnalisation du déploiement, nous avons pu effectuer l'installation d'une architecture DIET en tenant compte de serveurs NFS sur les clusters. Cela a permis de diminuer de façon significative le temps de déploiement. Le but de la dernière expérience est de montrer ce qu'apporte la décentralisation de l'administration au niveau de la reconfiguration. Nous avons comparé la reconfiguration effectuée à distance (un TUNe déployé sur une machine d'un cluster C reconfigure une entité logicielle installée sur un autre cluster différent de C) et la reconfiguration locale effectuée au sein du même cluster.

9.2 Perspectives

Dans la continuation des travaux présentés, deux aspects majeurs peuvent être poursuivis.

9.2.1 Tolérance aux pannes

- **Tolérance aux pannes des TUNe** : Le déploiement de plusieurs TUNe augmente la probabilité de pannes des systèmes d'administration. Lorsqu'un TUNe tombe en panne, l'administration de toutes les entités logicielles devient non fonctionnelle. Il est donc nécessaire de proposer de solutions pour remédier au problème de pannes de TUNe. Une solution consiste à introduire la notion de *sonde* au système TUNe et faire surveiller un nœuds dans la hiérarchie des TUNe par son père (la racine étant surveillée par l'un de ses fils). Avec cette approche lorsqu'un TUNe tombe en panne, son père (ou son fils) met en place des diagnostics pour le réparer.
- **Tolérance aux pannes lors du déploiement** : Parmi nos contributions, nous avons mis en œuvre la personnalisation du processus d'installation. L'exécution de ce processus peut engendrer des erreurs donc du dysfonctionnement du processus de déploiement. Il est à cet effet nécessaire de tenir compte des défaillances ou des erreurs qui surviennent durant le processus de déploiement. Dans ce cas de défaillances lors du déploiement d'une application TUNe peut par exemple être amené à *défaire* les opérations d'installation effectuées

(*rollback*) et signaler les erreurs produites, éventuellement consigner dans un rapport la réussite ou l'échec de l'installation. En cas d'échec, les raisons de l'échec sont explicitées : fichier source absent, protocole inaccessible, impossible de placer le fichier à destination (manque d'espace disque, permissions insuffisantes), etc.

9.2.2 La méta modélisation

Le formalisme de description utilisé par TUNe est basé sur le langage UML. Ce langage est initialement conçu pour représenter, spécifier et concevoir un système logiciel. L'utilisation du langage UML pour définir les politiques d'administration dans TUNe détourne l'usage premier du langage UML. Il apparaît donc plus judicieux de considérer que les langages permettant la définition de politiques d'administration dans TUNe sont des langages dédiés (des *Domain Specific Languages* ou *DSL*) à l'administration d'infrastructures logicielles. Il est alors possible de définir les méta-modèles de ces DSL, ce qui permet de définir précisément et formellement la syntaxe et la sémantique de ces langages. De plus, la définition de ces méta-modèles permet de construire des outils d'édition spécialisés grâce auxquels nous pourrons commencer la validation des diagrammes avant même le lancement de TUNe. Une proposition de méta-modèle a été publiée dans l'article suivant [CBTH08].

Quatrième partie

Annexe

Annexe A

Déploiement manuel d'une architecture DIET

A.1 Intrduction

Nous présentons dans cette annexe, comment déployer manuellement une architecture DIET. Les fichiers de configuration de chaque agent DIET est présenté avec une capture d'écran de lancement en ligne de commande. L'architecture DIET déployée est composée d'un omniNames (le serveur de nom), d'un MA (un Master Agent appelé MA1) relié à un LA (un Local Agent appelé LA1) qui est relié à un serveur SeD (Sever Daemon). Le résultat d'une dernière expérience de déploiement d'un système de monitoring (LeWYS) avec TUNe hiérarchisé est présenté à la fin de cette annexe.

A.1.1 Déploiement de serveur de nom : omniNames

La plate forme DIET est distribuée et composée de plusieurs agents(MA, LA, SeD). Toutes les communications internes sont assurées par un serveur de nom basé sur le bus CORBA. Ainsi tout le mécanisme de DIET repose sur lui, en particulier, les dialogues entres les agents (maîtres ou locaux). Pour utiliser DIET, il faut au préalable lancer l'exécutable du serveur de nom dénommé omniNames. Ce dernier nécessite un fichier de configuration et deux variables d'environnement. L'exemple d'un fichier de configuration est représenté par la figure A.1.

La première variable d'environnement (**LD_LIBRARY_PATH**) va indiquer le répertoire des librairies nécessaire au démarrage d'omniNames. La seconde variable (**OMNIORB_CONFIG**) indique le fichier de configuration. La figure A.2 montre le démarrage du serveur de nom.

```
#*******************************************************************************#
# (*) Reference to the CORBA Name server, to which all agents register and
#     connect to get references on other agent
#*******************************************************************************#

InitRef = NameService=corbaname::cleggs

InitRef = NameService=corbaname : :cleggs

#*******************************************************************************#
# giopMaxMsgSize: Max size of an omniORB message. It is useful if your client
#   has large amount of data to transfer onto the server. It defaults to 2MB.
#*******************************************************************************#

giopMaxMsgSize = 33554432 # 32MB

#*******************************************************************************#
# endPoint: Set the listening port of a DIET entity. It can be set in the
#   configuration file of this entity with a simpler syntax (ie, forces to
#   giop:tcp:localhost:<port>. Default is 2809, if available. But take a look
#   at the omniORB 4 User's Manual for more information.
#*******************************************************************************#

endPoint = giop:tcp::2809
```

FIGURE A.1 – Fichier de configuration d'omniNames

```
mtoure@cleggs:~/diet1.0$ export LD_LIBRARY_PATH=/home/mtoure/diet1.0/
mtoure@cleggs:~/diet1.0$ export OMNIORB_CONFIG=/home/mtoure/diet1.0/omniORB4.cfg
mtoure@cleggs:~/diet1.0$ ./omniNames -start -logdir /home/mtoure/diet1.0/dirLog

Thu Oct 15 09:29:06 2009:

Starting omniNames for the first time.
Wrote initial log file.
Read log file successfully.
Root context is IOR:010000002b00000049444c3a6f6d672e6f72672f436f734e616d696e672f4
00000003134372e3132372e3234302e31353200f90a00000b0000004e616d6553657276696365650002
0001000105090101000100000009010100
Checkpointing Phase 1: Prepare.
Checkpointing Phase 2: Commit.
Checkpointing completed.

Thu Oct 15 09:44:06 2009:

Checkpointing Phase 1: Prepare.
Checkpointing Phase 2: Commit.
Checkpointing completed.
```

FIGURE A.2 – Démarrage du serveur de nom omniNames

A présent, le service de nommage écoute sur le port par défaut (port 2809) et attend la connexion éventuelle d'un agent ou d'un client DIET.

A.1.2 Déploiement et configuration des agents DIET : MA, LA

Un fichier de configuration est nécessaire au lancement d'une entité DIET. On y trouve entre autre :

- le niveau de traçabilité ;
- le nom de l'agent maître ;

– le nom de l'agent local ;
– le lien avec un agent maître (s'il s'agit d'un agent local) ;
– le port d'écoute utilisé par les agents et les services DIET (port 2809).

Le démarrage d'un agent (MA ou LA) demande d'un fichier de configuration et les deux variables d'environnement précédemment décrites. Le figure suivantes montrent les fichier de configuration des agents DIET et une capture d'écran de démarrage d'un MA et d'un LA.

```
name = MA1
#******************************************************************************#
# dietPort: the listening port of the agent.
# dietHostname : the listening interface of the agent.
#******************************************************************************#
dietPort = 2809
dietHostname = cleggs.enseeiht.fr
#******************************************************************************#
# bindServicePort: port used by the Master Agent to share its IOR.
#******************************************************************************#
bindServicePort = 2001
#neighbours = hostma2:2001,hostma3:2001
#neighbours = <host:port>,<host:port>,<host:port>
minimumNeighbours = 2
maximumNeighbours = 10
updateLinkPeriod = 600
#******************************************************************************#
# fastUse: if set to 0, all LDAP and NWS parameters are ignored.
#******************************************************************************#
fastUse = 0
ldapUse  = 0
#ldapBase = cleggs:9050
#ldapMask = dc=LIP,dc=ens-lyon,dc=fr
#******************************************************************************#
# nwsUse:  0 tells FAST not to use NWS for its comm times forecasts.
#******************************************************************************#
nwsUse   = 0
nwsNameserver = cleggs:9056
#******************************************************************************#
# useLogService: 1 to use the LogService for monitoring.
#******************************************************************************#
useLogService = 0
lsOutbuffersize = 0
lsFlushinterval = 10000
```

FIGURE A.3 – Fichier de configuration d'un Master Agent MA

```
mtoure@cleggs:~/diet1.0$ export LD_LIBRARY_PATH=/home/mtoure/diet1.0/
mtoure@cleggs:~/diet1.0$ export OMNIORB_CONFIG=/home/mtoure/diet1.0/omniORB4.cfg
mtoure@cleggs:~/diet1.0$ ./dietAgent /home/mtoure/diet1.0/MA1.cfg

Master Agent MA1 started.
An agent has registered from << cleggs, with 1 services.

**************************************************
Got request 0 on problem MatSUM
Got a response from 0th child to request 0
**************************************************

**************************************************
Got request 1 on problem MatSUM
Got a response from 0th child to request 1
**************************************************
```

FIGURE A.4 – Démarrage d'un agent MA

```
name = LA1
#***********************************************************************#
# (*) parentName: the name of the agent to which the LA will register. This
#***********************************************************************#
parentName = MA1
#***********************************************************************#
# dietHostname : the listening interface of the agent.
#***********************************************************************#
dietPort = 2809
dietHostname = cleggs.enseeiht.fr
#***********************************************************************#
# bindServicePort: port used by the Master Agent to share its IOR.
#***********************************************************************#
bindServicePort = 2001
#neighbours = hostma2:2001,hostma3:2001
#neighbours = <host:port>,<host:port>,<host:port>
minimumNeighbours = 2
maximumNeighbours = 10
updateLinkPeriod = 600
#***********************************************************************#
# fastUse: if set to 0, all LDAP and NWS parameters are ignored.
#***********************************************************************#
fastUse = 0
ldapUse  = 0
#ldapBase = cleggs:9050
#***********************************************************************#
# nwsUse:  0 tells FAST not to use NWS for its comm times forecasts.
#***********************************************************************#
nwsUse   = 0
nwsNameserver = cleggs:9056
#***********************************************************************#
# useLogService: 1 to use the LogService for monitoring.
#***********************************************************************#
useLogService = 0
lsOutbuffersize = 0
lsFlushinterval = 10000
```

FIGURE A.5 – Fichier de configuration d'un Local Agent LA

```
mtoure@cleggs:~/diet1.0$ export LD_LIBRARY_PATH=/home/mtoure/diet1.0/
mtoure@cleggs:~/diet1.0$ export OMNIORB_CONFIG=/home/mtoure/diet1.0/omniORB4.cfg
mtoure@cleggs:~/diet1.0$ ./dietAgent /home/mtoure/diet1.0/LA1.cfg

Local Agent LA1 started.
A server has registered from cleggs, with 1 services.
DIET INTERNAL WARNING: too many parameters for NWS scheduler (0,3,2,1), reverting to default.
This is not a fatal bug, but please send a report to diet-dev@ens-lyon.fr

**************************************************
Got request 0 on problem MatSUM
Got a response from 0th child to request 0
DIET INTERNAL WARNING: too many parameters for NWS scheduler (0,3,2,1), reverting to default.
This is not a fatal bug, but please send a report to diet-dev@ens-lyon.fr

**************************************************
Got request 1 on problem MatSUM
Got a response from 0th child to request 1
```

FIGURE A.6 – Démarrage d'un agent LA

A.1.3 Déploiement des serveurs : SeD

Pour lancer un serveur DIET(SeD) il faut exécuter un programme serveur qui ira s'enregistrer auprès de l'architecture d'omniNames. Des exemples de calcul matriciel sont fournis avec DIET. Pour les tests on a utilisé l'exemple dmat-manips qui contient entre autre le binaire server et des binaires clients. Pour lancer le SeD il faut lui passer en paramètre un fichier de configuration dont un exemple est présente sur la figure suivante.

```
1
2  #****************************************************************************#
3  # traceLevel for the DIET SeD library:
4  #    0  DIET prints only warnings and errors on the standard error output.
5  #    1  [default] DIET prints information on the main steps of a call.
6  #    5  DIET prints information on all internal steps too.
7  #   10  DIET prints all the communication structures too.
8  #  >10  (traceLevel - 10) is given to the ORB to print CORBA messages too.
9  #****************************************************************************#
10 traceLevel = 1
11 #****************************************************************************#
12 # (*) parentName: the name of the agent to which the SeD will register. This
13 #   agent must have registered at the same CORBA Naming Service that is
14 #   pointed to by your ORB configuration.
15 #****************************************************************************#
16 parentName = LA1
17 dietPort = 2809
18 dietHostname = cleggs.enseeiht.fr
19 fastUse = 0
20 ldapUse  = 1
21 ldapBase = cleggs:9050
22 #****************************************************************************#
23 # nwsUse:  0 tells FAST not to use NWS for its comm times forecasts.
24 # nwsNameserver: <host:port> of the NWS nameserver.
25 #****************************************************************************#
26 nwsUse   = 1
27 nwsNameserver = cleggs:9056
28 #****************************************************************************#
29 # useLogService:    1 to use the LogService for monitoring.
30 # lsOutbuffersize:  the size of the buffer for outgoing messages.
31 # lsFlushinterval:  the flush interval for the outgoing message buffer.
32 #****************************************************************************#
33 useLogService = 0
34 lsOutbuffersize = 0
35 lsFlushinterval = 10000
```

FIGURE A.7 – Fichier de configuration d'un SeD

A.1.4 Déploiement de LeWYS

Monitoring is at the heart of cluster or grid management. Instrumentation data is used to schedule tasks, load-balance devices and services, notify administrators of hardware and software failures, and generally monitor the health and usage of a system.

LeWYS (LeWYS is Watching Your System) [CELQ04] is a component-based framework for distributed system monitoring and aimed at monitoring grid. It relies on the use of monitor daemons (*Probe*) and *observers*. A *Probe*, running on every monitored node, probes a fixed set of hardware resources at a fixed sample rate and multicasts the values on the network *observers*. *observers* collects data from multiple LeWYS Monitoring Probe sources and saves all data in files.

Figure A.9 gives the results on 1032 grid nodes for 1, 2 and 5 TUNe. It shows that use of 5 TUNe to deploy 1200 probes on 1032 nodes of Grid'5000 has deceased the deployement execution time.

```
mtoure@cleggs:~/diet1.0$ export LD_LIBRARY_PATH=/home/mtoure/diet1.0/
mtoure@cleggs:~/diet1.0$ export OMNIORB_CONFIG=/home/mtoure/diet1.0/omniORB4.cfg
mtoure@cleggs:~/diet1.0$ ./server /home/mtoure/diet1.0/SeD1.cfg MatSUM

------------------------------------------------
Service Table (1 services)
------------------------------------------------

- Service MatSUM
     IN    matrix of double
     IN    matrix of double
     OUT   matrix of double
 - Convertor to base/plus
     IN    NB_ROW of argument 0 (out: 0)
     IN    NB_COL of argument 0 (out: 0)
     IN    IDENT  of argument 0 (out: 0)
     IN    IDENT  of argument 1 (out: 1)
     OUT   IDENT  of argument 2 (out: 2)

**************************************************
Got request 0

SeD::solve invoked on pb: MatSUM
Calling getSolver
Solve MatSUM ... done
SeD::solve complete
****************************************************************

**************************************************
Got request 1

SeD::solve invoked on pb: MatSUM
Calling getSolver
Solve MatSUM ... done
SeD::solve complete
****************************************************************
```

FIGURE A.8 – Démarrage d'un serveur qui effectue la somme matricielle

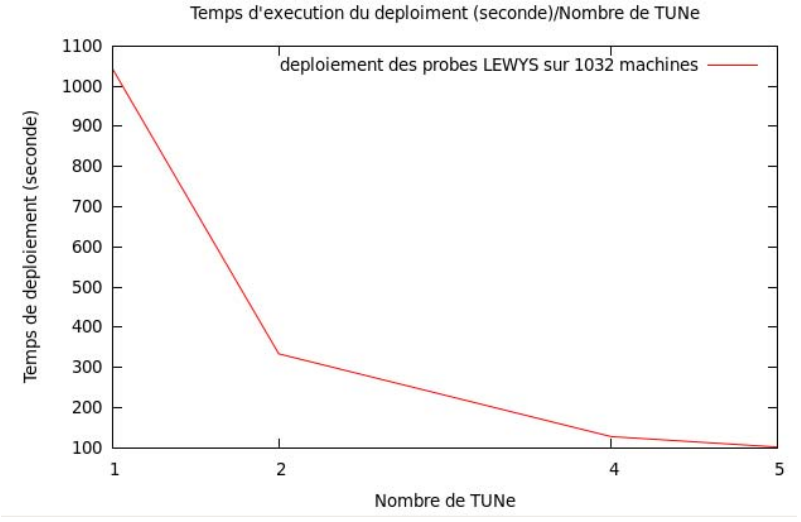

FIGURE A.9 – *Hierarchical deployment result whith LeWYS*

Bibliographie

[ADZ00] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves : A mechanism for resource management in cluster-based network servers. In *In Proceedings of the ACM SIGMETRICS Conference*, pages 90–101, 2000.

[BBH+05] Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sacha Krakowiak, Adrian Mos, Noel De Palma, Vivien Quéma, and Jean-Bernard Stefani. Architecture-based autonomous repair management : An application to j2ee clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS-2005). Orlando, FL*, 2005.

[BCL+06a] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12) :1257–1284, 2006.

[BCL+06b] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. In *Software - Practice and Experience, special issue on "Experiences with Auto-adaptive and Reconfigurable Systems", 36(11-12) :1257-1284*, 2006.

[BHS+08] Laurent Broto, Daniel Hagimont, Patricia Stolf, Noel Depalma, and Suzy Temate. Autonomic management policy specification in tune. In *SAC '08 : Proceedings of the 2008 ACM symposium on Applied computing*, pages 1658–1663, New York, NY, USA, 2008. ACM.

[Bra0x] T. Bray. Extensible markup language (xml). 200x. http://www.w3.org/pub/WWW/TR/WD-xml-lang.html.

[Bro08] Laurent Broto. *Support langage et système pour l'administration autonome*. Thèse de doctorat, Université de Toulouse, Toulouse, France, septembre 2008.

[BSB+08] Laurent Broto, Patricia Stolf, Jean-Paul Bahsoun, Daniel Hagimont, and N. Depalma. Spécification de politiques d'administration autonome avec Tune. In *Conférence Française sur les Systèmes d'Exploitation (CFSE), Fribourg, Suisse, 11/02/08-13/02/08*, page (support électronique), http ://www.sigops-france.fr/, février 2008. Chapitre Français de l'ACM SIGOPS.

[Buy99] Rajkumar Buyya. *High Performance Cluster Computing : Architectures and Systems, Vol. 1*. Prentice Hall PTR, 1 edition, May 1999.

[CBTH08] Benoit Combemale, Laurent Broto, Alain Bouzaide Tchana, and Daniel Hagimont. Metamodeling Autonomic System Management Policies –

Ongoing Works. In *IEEE International Workshop on Model-Driven Development of Autonomic Systems , Turku, Finland, 28/07/08-01/08/08*, http ://www.ieee.org/, juillet 2008. IEEE.

[CC95] Thierry Coupaye and Christine Collet. Denotational semantics for an active rule execution model. In *RIDS '95 : Proceedings of the Second International Workshop on Rules in Database Systems*, pages 36–50, London, UK, 1995. Springer-Verlag.

[CCD+05a] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000 : A large scale and highly reconfigurable grid experimental testbed. In *GRID '05 : Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society.

[CCD06] Eddy Caron, Pushpinder Kaur Chouhan, and Holly Dail. Godiet : a deployment tool for distributed middleware on grid'5000. 2006.

[CCG+05] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. *CoRR*, abs/cs/0506006, 2005.

[CD06] Eddy Caron and Frédéric Desprez. Diet : A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3) :335–352, 2006.

[CELQ04] Emmanuel Cecchet, Hazem Elmeleegy, Oussama Layaida, and Vivien Quéma. Implementing probes for j2ee cluster monitoring. In *In OOPSLA Workshop on Component and Middleware Proformance*, 2004.

[CFH+98] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, Dennis Heimbigner, André van der Hoek, Alexander L. Wolf, Andre Van Der, Er L. Wolf, and Er L. Wolf. A characterization framework for software deployment technologies. Technical report, 1998.

[CHR09] Benoit Claudel, Guillaume Huard, and Olivier Richard. Taktuk, adaptive deployment of remote executions. In *HPDC '09 : Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 91–100, New York, NY, USA, 2009. ACM.

[CIG+03] Jeffrey S. Chase, David E. Irwin, Laura E. Grit, Justin D. Moore, and Sara E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *HPDC '03 : Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, page 90, Washington, DC, USA, 2003. IEEE Computer Society.

[CLM05] Pierre-Yves Cunin, Vincent Lestideau, and Noélle Merle. Orya : A strategy oriented deployment framework. In *Component Deployment*, pages 177–180, 2005.

[CLQS02] Philippe Combes, Frédéric Lombard, Martin Quinson, and Frédéric Suter. A scalable approach to network enabled servers. In *ASIAN '02 : Proceedings of the7th Asian Computing Science Conference on Advances in Computing Science*, pages 110–124, London, UK, 2002. Springer-Verlag.

[DFDM08] Jérémy Dubus, Areski Flissi, Nicolas Dolet, and Philippe Merle. Une démarche orientée modèle pour déployer des systèmes logiciels répartis. *L'OBJET*, 14(1-2) :35–59, 2008.

[FDDM08] Areski Flissi, Jérémy Dubus, Nicolas Dolet, and Philippe Merle. Deploying on the grid with deployware. In *CCGRID '08 : Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 177–184, Washington, DC, USA, 2008. IEEE Computer Society.

[FFG+01] S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Oceano - sla based management of a computing utility. In *In Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, pages 855–868, 2001.

[FM06] Areski Flissi and Philippe Merle. A generic deployment framework for grid computing and distributed applications. In *International Symposium on Grid computing, high-performAnce and Distributed Applications (GADA'06)*, Montpellier, France, nov 2006.

[GC03] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1) :5–18, 2003.

[GGL+03] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. Smartfrog : Configuration and automatic ignition of distributed applications. Technical report, HP, 2003.

[gra0xa] graal. Logservice. 200x. http://graal.ens-lyon.fr/DIET/logservice.html.

[gra0xb] graal. xml godiet generator. 200x. http://graal.ens-lyon.fr/~diet/xmlgodietgenerator.html.

[GTLAG05] Bhavin Gandhi, Sameer Tilak, Michael J. Lewis, and Nael B. Abu-Ghazaleh. Controlling the coverage of grid information dissemination protocols. In *NCA '05 : Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, pages 267–270, Washington, DC, USA, 2005. IEEE Computer Society.

[Hal99] Richard Scott Hall. *Agent-based software configuration and deployment*. PhD thesis, Boulder, CO, USA, 1999. Director-Wolf, Alexander L.

[HHW99] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A cooperative approach to support software deployment using the software dock. In *ICSE '99 : Proceedings of the 21st international conference on Software engineering*, pages 174–183, New York, NY, USA, 1999. ACM.

[Hor01] Paul Horn. Autonomic computing : Ibm's perspective on the state of information technology, 2001.

[JCC+03] Yun Fu Jeffrey, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. Sharp : An architecture for secure resource peering. In *In Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 133–148, 2003.

[Kep05] Jeffrey O. Kephart. Research challenges of autonomic computing. In *ICSE '05 : Proceedings of the 27th international conference on Software engineering*, pages 15–22, New York, NY, USA, 2005. ACM.

[KF98b] Carl Kesselman and Ian Foster. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[Kie04] Thilo Kielmann. A hierarchy of network performance characteristics for grid applications and services. 2004.

[Kle88] S.M. Klerer. The osi management architecture : an overview. *Network, IEEE*, 2(2) :20–29, Mar 1988.

[Lac05] Sébastien Lacour. *Contribution à l'automatisation du déploiement d'applications sur des grilles de calcul*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, December 2005. in French.

[LB03] Vincent Lestideau and Noureddine Belkhatir. Providing highly automated and generic means for. In *Process. European Workshop on Software Process Technology*, pages 1–2, 2003.

[LGWT06] Hui Li, David Groep, Lex Wolters, and Jeff Templon. Job failure analysis and its implications in a large-scale production grid. In *E-SCIENCE '06 : Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 27, Washington, DC, USA, 2006. IEEE Computer Society.

[LPP04b] Sebastien Lacour, Christian Perez, and Thierry Priol. A network topology description model for grid application deployment. In *GRID '04 : Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 61–68, Washington, DC, USA, 2004. IEEE Computer Society.

[LPP05] S. Lacour, C. Perez, and T. Priol. Generic application description model : Toward automatic deployment of applications on computational grids. In *GRID '05 : Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 284–287, Washington, DC, USA, 2005. IEEE Computer Society.

[Mar04] Cyrille Martin. Déploiement et contrôle d'applications parallèles sur grappes de grandes tailles. 2004.

[MB04] Noélle Merle and Noureddine Belkhatir. Une architecture conceptuelle pour le déploiement d'applications à grande échelle. In *INFORSID*, pages 461–476, 2004.

[MCBS03] R. Medeiros, W. Cirne, F. Brasileiro, and J. Sauve. Faults in grids : why are they so bad and what can be done about it ? In *Grid Computing, 2003. Proceedings. Fourth International Workshop on*, pages 18–24, Nov. 2003.

[MR03] Cyrille Martin and O. Richard. Algorithme de vol de travail appliqué au déploiement d'applications parallèles. In *Actes Renpar 15*, 2003.

[Mur04] Richard Murch. *Autonomic Computing*. IBM Press, 2004.

[OAR0x] OARGrid. Oargrid project. 200x. http://gforge.inria.fr/projects/oargrid/.

[PPR02] Christian Pérez, Thierry Priol, and André Ribes. A parallel corba component model for numerical code coupling. In *GRID '02 : Proceedings of the Third International Workshop on Grid Computing*, pages 88–99, London, UK, 2002. Springer-Verlag.

[RM07b] Romain Rouvoy and Philippe Merle. Un langage de description et de vérification de motifs d'architecture : Fractal adl. In *LMO*, pages 49–64, 2007.

[Sab06] Ritu Sabharwal. Grid infrastructure deployment using smartfrog technology. In *ICNS '06 : Proceedings of the International conference on Networking and Services*, page 73, Washington, DC, USA, 2006. IEEE Computer Society.

[SBDP08] Sylvain Sicard, Fabienne Boyer, and Noel De Palma. Using components for architecture-based management : the self-repair case. In *ICSE '08 : Proceedings of the 30th international conference on Software engineering*, pages 101–110, New York, NY, USA, 2008. ACM.

[SLAS07] Julien Sopena, Fabrice Legond, Luciana Arantes, and Pierre Sens. A composition approach to mutual exclusion algorithms for grid applications. In *The 36th International Conference on Parallel Processing (ICPP07 - XiAn, CN)*, pages 65–75. IEEE Computer Society, September 2007.

[TBD+06] Christophe Taton, Sara Bouchenak, Noel Depalma, Daniel Hagimont, and Sacha Krakowiak. Administration autonome de services Internet : Expérience avec l'auto-optimisation. In *Conférence Française sur les Systèmes d'Exploitation (CFSE), Canet en Roussillon, 04/10/06-06/10/06*, page (support électronique), http ://www.sigops-france.fr/, octobre 2006. Chapitre Français de l'ACM SIGOPS.

[TBDP+06] Christophe Taton, Sara Bouchenak, Noel De Palma, Daniel Hagimont, and Sylvain Sicard. Self-sizing of clustered databases. In *WOWMOM '06 : Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, pages 506–512, Washington, DC, USA, 2006. IEEE Computer Society.

[TBS+08] Mahamadou Toure, Girma Berhe, Patricia Stolf, Laurent Broto, N. Depalma, and Daniel Hagimont. Autonomic Management for Grid Applications. In *Euromicro International Conference on Parallel, Distributed and network-based Processing, Toulouse, 13/02/08-15/02/08*, pages 79–86, http ://www.ieee.org/, février 2008. IEEE.

[TH09] Mahamadou Toure and Daniel Hagimont. Déploiement à grande échelle. In *Conférence Française sur les Systèmes d'Exploitation (CFSE), Toulouse, 09/09/09-11/09/09*, page (support électronique), 2009.

[TSHB10] Mahamadou Toure, Patricia Stolf, Daniel Hagimont, and Laurent Broto. Large scale deployment (regular paper). In *International Conference on Autonomic and Autonomous Systems (ICAS), Cancun, Mexico, 07/03/10-13/03/10*, page (electronic medium), http ://www.ieee.org/, 2010. IEEE.

[WGR96] Anthony S KJELLUM William G ROPP, Ewing L USK. Using mpi-portable parallel programming with the message-passing interface, by william gropp. *Sci. Program.*, 5(3) :275–276, 1996.

Liste des tableaux

Table des figures